WILEY

Digital Logic Testing and Simulation

Second Edition

ALEXANDER MICZO

VISIT…

DIGITAL LOGIC TESTING AND SIMULATION

DIGITAL LOGIC TESTING AND SIMULATION

SECOND EDITION

Alexander Miczo

A JOHN WILEY & SONS, INC., PUBLICATION

Copyright © 2003 by John Wiley & Sons, Inc. All rights reserved.

Published by John Wiley & Sons, Inc., Hoboken, New Jersey.
Published simultaneously in Canada.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning, or otherwise, except as permitted under Section 107 or 108 of the 1976 United States Copyright Act, without either the prior written permission of the Publisher, or authorization through payment of the appropriate per-copy fee to the Copyright Clearance Center, Inc., 222 Rosewood Drive, Danvers, MA 01923, 978-750-8400, fax 978-750-4470, or on the web at www.copyright.com. Requests to the Publisher for permission should be addressed to the Permissions Department, John Wiley & Sons, Inc., 111 River Street, Hoboken, NJ 07030, (201) 748-6011, fax (201) 748-6008, e-mail: permreq@wiley.com.

Limit of Liability/Disclaimer of Warranty: While the publisher and author have used their best efforts in preparing this book, they make no representations or warranties with respect to the accuracy or completeness of the contents of this book and specifically disclaim any implied warranties of merchantability or fitness for a particular purpose. No warranty may be created or extended by sales representatives or written sales materials. The advice and strategies contained herein may not be suitable for your situation. You should consult with a professional where appropriate. Neither the publisher nor author shall be liable for any loss of profit or any other commercial damages, including but not limited to special, incidental, consequential, or other damages.

For general information on our other products and services please contact our Customer Care Department within the U.S. at 877-762-2974, outside the U.S. at 317-572-3993 or fax 317-572-4002.

Wiley also publishes its books in a variety of electronic formats. Some content that appears in print, however, may not be available in electronic format.

Library of Congress Cataloging-in-Publication Data:

Miczo, Alexander.
Digital logic testing and simulation / Alexander Miczo—2nd ed.
p. cm.
Rev. ed. of: Digital logic testing and simulation. c1986.
Includes bibliographical references and index.
ISBN 0-471-43995-9 (cloth)
1. Digital electronics—Testing. I. Miczo, Alexander. Digital logic testing and simulation
II. Title.

TK7868.D5M49 2003
621.3815′48—dc21

2003041100

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

CONTENTS

Preface **xvii**

1 Introduction **1**

1.1 Introduction 1

1.2 Quality 2

1.3 The Test 2

1.4 The Design Process 6

1.5 Design Automation 9

1.6 Estimating Yield 11

1.7 Measuring Test Effectiveness 14

1.8 The Economics of Test 20

1.9 Case Studies 23

1.9.1 The Effectiveness of Fault Simulation 23

1.9.2 Evaluating Test Decisions 24

1.10 Summary 26

Problems 29

References 30

2 Simulation **33**

2.1 Introduction 33

2.2 Background 33

2.3 The Simulation Hierarchy 36

2.4 The Logic Symbols 37

2.5 Sequential Circuit Behavior 39

2.6 The Compiled Simulator 44

2.6.1 Ternary Simulation 48

2.6.2 Sequential Circuit Simulation 48
2.6.3 Timing Considerations 50
2.6.4 Hazards 50
2.6.5 Hazard Detection 52
2.7 Event-Driven Simulation 54
2.7.1 Zero-Delay Simulation 56
2.7.2 Unit-Delay Simulation 58
2.7.3 Nominal-Delay Simulation 59
2.8 Multiple-Valued Simulation 61
2.9 Implementing the Nominal-Delay Simulator 64
2.9.1 The Scheduler 64
2.9.2 The Descriptor Cell 67
2.9.3 Evaluation Techniques 70
2.9.4 Race Detection in Nominal-Delay Simulation 71
2.9.5 Min–Max Timing 72
2.10 Switch-Level Simulation 74
2.11 Binary Decision Diagrams 86
2.11.1 Introduction 86
2.11.2 The Reduce Operation 91
2.11.3 The Apply Operation 96
2.12 Cycle Simulation 101
2.13 Timing Verification 106
2.13.1 Path Enumeration 107
2.13.2 Block-Oriented Analysis 108
2.14 Summary 110
Problems 111
References 116

3 Fault Simulation 119

3.1 Introduction 119
3.2 Approaches to Testing 120
3.3 Analysis of a Faulted Circuit 122
3.3.1 Analysis at the Component Level 122
3.3.2 Gate-Level Symbols 124
3.3.3 Analysis at the Gate Level 124

3.4 The Stuck-At Fault Model 125
3.4.1 The AND Gate Fault Model 127
3.4.2 The OR Gate Fault Model 128
3.4.3 The Inverter Fault Model 128
3.4.4 The Tri-State Fault Model 128
3.4.5 Fault Equivalence and Dominance 129
3.5 The Fault Simulator: An Overview 131
3.6 Parallel Fault Processing 134
3.6.1 Parallel Fault Simulation 134
3.6.2 Performance Enhancements 136
3.6.3 Parallel Pattern Single Fault Propagation 137
3.7 Concurrent Fault Simulation 139
3.7.1 An Example of Concurrent Simulation 139
3.7.2 The Concurrent Fault Simulation Algorithm 141
3.7.3 Concurrent Fault Simulation: Further Considerations 146
3.8 Delay Fault Simulation 147
3.9 Differential Fault Simulation 149
3.10 Deductive Fault Simulation 151
3.11 Statistical Fault Analysis 152
3.12 Fault Simulation Performance 155
3.13 Summary 157
Problems 159
References 162

4 Automatic Test Pattern Generation 165

4.1 Introduction 165
4.2 The Sensitized Path 165
4.2.1 The Sensitized Path: An Example 166
4.2.2 Analysis of the Sensitized Path Method 168
4.3 The D-Algorithm 170
4.3.1 The D-Algorithm: An Analysis 171
4.3.2 The Primitive D-Cubes of Failure 174
4.3.3 Propagation D-Cubes 177
4.3.4 Justification and Implication 179
4.3.5 The D-Intersection 180

4.4 Testdetect 182
4.5 The Subscripted D-Algorithm 184
4.6 PODEM 188
4.7 FAN 193
4.8 Socrates 202
4.9 The Critical Path 205
4.10 Critical Path Tracing 208
4.11 Boolean Differences 210
4.12 Boolean Satisfiability 216
4.13 Using BDDs for ATPG 219
4.13.1 The BDD XOR Operation 219
4.13.2 Faulting the BDD Graph 220
4.14 Summary 224
Problems 226
References 230

5 Sequential Logic Test 233

5.1 Introduction 233
5.2 Test Problems Caused by Sequential Logic 233
5.2.1 The Effects of Memory 234
5.2.2 Timing Considerations 237
5.3 Sequential Test Methods 239
5.3.1 Seshu's Heuristics 239
5.3.2 The Iterative Test Generator 241
5.3.3 The 9-Value ITG 246
5.3.4 The Critical Path 249
5.3.5 Extended Backtrace 250
5.3.6 Sequential Path Sensitization 252
5.4 Sequential Logic Test Complexity 259
5.4.1 Acyclic Sequential Circuits 260
5.4.2 The Balanced Acyclic Circuit 262
5.4.3 The General Sequential Circuit 264
5.5 Experiments with Sequential Machines 266
5.6 A Theoretical Limit on Sequential Testability 272

5.7 Summary 277

Problems 278

References 280

6 Automatic Test Equipment 283

6.1 Introduction 283

6.2 Basic Tester Architectures 284
6.2.1 The Static Tester 284
6.2.2 The Dynamic Tester 286

6.3 The Standard Test Interface Language 288

6.4 Using the Tester 293

6.5 The Electron Beam Probe 299

6.6 Manufacturing Test 301

6.7 Developing a Board Test Strategy 304

6.8 The In-Circuit Tester 307

6.9 The PCB Tester 310
6.9.1 Emulating the Tester 311
6.9.2 The Reference Tester 312
6.9.3 Diagnostic Tools 313

6.10 The Test Plan 315

6.11 Visual Inspection 316

6.12 Test Cost 319

6.13 Summary 319

Problems 320

References 321

7 Developing a Test Strategy 323

7.1 Introduction 323

7.2 The Test Triad 323

7.3 Overview of the Design and Test Process 325

7.4 A Testbench 327
7.4.1 The Circuit Description 327
7.4.2 The Test Stimulus Description 330

7.5 Fault Modeling 331
7.5.1 Checkpoint Faults 331
7.5.2 Delay Faults 333
7.5.3 Redundant Faults 334
7.5.4 Bridging Faults 335
7.5.5 Manufacturing Faults 337
7.6 Technology-Related Faults 337
7.6.1 MOS 338
7.6.2 CMOS 338
7.6.3 Fault Coverage Results in Equivalent Circuits 340
7.7 The Fault Simulator 341
7.7.1 Random Patterns 342
7.7.2 Seed Vectors 343
7.7.3 Fault Sampling 346
7.7.4 Fault-List Partitioning 347
7.7.5 Distributed Fault Simulation 348
7.7.6 Iterative Fault Simulation 348
7.7.7 Incremental Fault Simulation 349
7.7.8 Circuit Initialization 349
7.7.9 Fault Coverage Profiles 350
7.7.10 Fault Dictionaries 351
7.7.11 Fault Dropping 352
7.8 Behavioral Fault Modeling 353
7.8.1 Behavioral MUX 354
7.8.2 Algorithmic Test Development 356
7.8.3 Behavioral Fault Simulation 361
7.8.4 Toggle Coverage 364
7.8.5 Code Coverage 365
7.9 The Test Pattern Generator 368
7.9.1 Trapped Faults 368
7.9.2 SOFTG 369
7.9.3 The Imply Operation 369
7.9.4 Comprehension Versus Resolution 371
7.9.5 Probable Detected Faults 372
7.9.6 Test Pattern Compaction 372
7.9.7 Test Counting 374
7.10 Miscellaneous Considerations 378
7.10.1 The ATPG/Fault Simulator Link 378

7.10.2 ATPG User Controls 380
7.10.3 Fault-List Management 381
7.11 Summary 382
Problems 383
References 385

8 Design-For-Testability 387

8.1 Introduction 387
8.2 Ad Hoc Design-for-Testability Rules 388
8.2.1 Some Testability Problems 389
8.2.2 Some Ad Hoc Solutions 393
8.3 Controllability/Observability Analysis 396
8.3.1 SCOAP 396
8.3.2 Other Testability Measures 403
8.3.3 Test Measure Effectiveness 405
8.3.4 Using the Test Pattern Generator 406
8.4 The Scan Path 407
8.4.1 Overview 407
8.4.2 Types of Scan-Flops 410
8.4.3 Level-Sensitive Scan Design 412
8.4.4 Scan Compliance 416
8.4.5 Scan-Testing Circuits with Memory 418
8.4.6 Implementing Scan Path 420
8.5 The Partial Scan Path 426
8.6 Scan Solutions for PCBs 432
8.6.1 The NAND Tree 433
8.6.2 The 1149.1 Boundary Scan 434
8.7 Summary 443
Problems 444
References 449

9 Built-In Self-Test 451

9.1 Introduction 451
9.2 Benefits of BIST 452
9.3 The Basic Self-Test Paradigm 454

9.3.1 A Mathematical Basis for Self-Test 455
9.3.2 Implementing the LFSR 459
9.3.3 The Multiple Input Signature Register (MISR) 460
9.3.4 The BILBO 463
9.4 Random Pattern Effectiveness 464
9.4.1 Determining Coverage 464
9.4.2 Circuit Partitioning 465
9.4.3 Weighted Random Patterns 467
9.4.4 Aliasing 470
9.4.5 Some BIST Results 471
9.5 Self-Test Applications 471
9.5.1 Microprocessor-Based Signature Analysis 471
9.5.2 Self-Test Using MISR/Parallel SRSG (STUMPS) 474
9.5.3 STUMPS in the ES/9000 System 477
9.5.4 STUMPS in the S/390 Microprocessor 478
9.5.5 The Macrolan Chip 480
9.5.6 Partial BIST 482
9.6 Remote Test 484
9.6.1 The Test Controller 484
9.6.2 The Desktop Management Interface 487
9.7 Black-Box Testing 488
9.7.1 The Ordering Relation 489
9.7.2 The Microprocessor Matrix 493
9.7.3 Graph Methods 494
9.8 Fault Tolerance 495
9.8.1 Performance Monitoring 496
9.8.2 Self-Checking Circuits 498
9.8.3 Burst Error Correction 499
9.8.4 Triple Modular Redundancy 503
9.8.5 Software Implemented Fault Tolerance 505
9.9 Summary 505
Problems 507
References 510

10 Memory Test 513

10.1 Introduction 513

10.2 Semiconductor Memory Organization 514
10.3 Memory Test Patterns 517
10.4 Memory Faults 521
10.5 Memory Self-Test 524
10.5.1 A GALPAT Implementation 525
10.5.2 The 9N and 13N Algorithms 529
10.5.3 Self-Test for BIST 531
10.5.4 Parallel Test for Memories 531
10.5.5 Weak Read–Write 533
10.6 Repairable Memories 535
10.7 Error Correcting Codes 537
10.7.1 Vector Spaces 538
10.7.2 The Hamming Codes 540
10.7.3 ECC Implementation 542
10.7.4 Reliability Improvements 543
10.7.5 Iterated Codes 545
10.8 Summary 546
Problems 547
References 549

11 I_{DDQ} 551

11.1 Introduction 551
11.2 Background 551
11.3 Selecting Vectors 553
11.3.1 Toggle Count 553
11.3.2 The Quietest Method 554
11.4 Choosing a Threshold 556
11.5 Measuring Current 557
11.6 I_{DDQ} Versus Burn-In 559
11.7 Problems with Large Circuits 562
11.8 Summary 564
Problems 565
References 565

12 Behavioral Test and Verification 567

12.1 Introduction 567

12.2 Design Verification: An Overview 568

12.3 Simulation 570
12.3.1 Performance Enhancements 570
12.3.2 HDL Extensions and C++ 572
12.3.3 Co-design and Co-verification 573

12.4 Measuring Simulation Thoroughness 575
12.4.1 Coverage Evaluation 575
12.4.2 Design Error Modeling 578

12.5 Random Stimulus Generation 581

12.6 The Behavioral ATPG 587
12.6.1 Overview 587
12.6.2 The RTL Circuit Image 588
12.6.3 The Library of Parameterized Modules 589
12.6.4 Some Basic Behavioral Processing Algorithms 593

12.7 The Sequential Circuit Test Search System (SCIRTSS) 597
12.7.1 A State Traversal Problem 597
12.7.2 The Petri Net 602

12.8 The Test Design Expert 607
12.8.1 An Overview of TDX 607
12.8.2 DEPOT 614
12.8.3 The Fault Simulator 616
12.8.4 Building Goal Trees 617
12.8.5 Sequential Conflicts in Goal Trees 618
12.8.6 Goal Processing for a Microprocessor 620
12.8.7 Bidirectional Goal Search 624
12.8.8 Constraint Propagation 625
12.8.9 Pitfalls When Building Goal Trees 626
12.8.10 MaxGoal Versus MinGoal 627
12.8.11 Functional Walk 629
12.8.12 Learn Mode 630
12.8.13 DFT in TDX 633

12.9 Design Verification 635
12.9.1 Formal Verification 636
12.9.2 Theorem Proving 636

12.9.3 Equivalence Checking 638
12.9.4 Model Checking 640
12.9.5 Symbolic Simulation 648
12.10 Summary 650
Problems 652
References 653
Index **657**

PREFACE

About one and a half decades ago the state of the art in DRAMs was 64K bytes, a typical personal computer (PC) was implemented with about 60 to 100 dual in-line packages (DIPs), and the VAX11/780 was a favorite platform for electronic design automation (EDA) developers. It delivered computational power rated at about one MIP (million instructions per second), and several users frequently shared this machine through VT100 terminals.

Now, CPU performance and DRAM capacity have increased by more than three orders of magnitude. The venerable VAX11/780, once a benchmark for performance comparison and host for virtually all EDA programs, has been relegated to museums, replaced by vastly more powerful PCs, implemented with fewer than a half dozen integrated circuits (ICs), at a fraction of the cost. Experts predict that shrinking geometries, and resultant increase in performance, will continue for at least another 10 to 15 years.

Already, it is becoming a challenge to use the available real estate on a die. Whereas in the original Pentium design various teams vied for a few hundred additional transistors on the die,[1] it is now becoming increasingly difficult for a design team to use all of the available transistors.[2]

The ubiquitous 8-bit microcontroller appears in entertainment products and in automobiles; billions are sold each year. Gordon Moore, Chairman Emeritus of Intel Corp., observed that these less glamorous workhorses account for more than 98% of Intel's unit sales.[3] More complex ICs perform computation, control, and communications in myriad applications. With contemporary EDA tools, one logic designer can create complex digital designs that formerly required a team of a half dozen logic designers or more. These tools place logic design capability into the hands of an ever-growing number of users. Meanwhile, these development tools themselves continue to evolve, reducing turn-around time from design of logic circuit to receipt of fabricated parts.

This rapid advancement is not without problems. Digital test and verification present major hurdles to continued progress. Problems associated with digital logic testing have existed for as long as digital logic itself has existed. However, these problems have been exacerbated by the growing number of circuits on individual chips. One development group designing a RISC (reduced instruction set computer) stated,[4] "the work required to ... test a chip of this size approached the amount of effort required to design it. If we had started over, we would have used more resources on this tedious but important chore."

The increase in size and complexity of circuits on a chip, often with little or no increase in the number of I/O pins, creates a testing bottleneck. Much more logic must be controlled and observed with the same number of I/O pins, making it more difficult to test the chip. Yet, the need for testing continues to grow in importance. The test must detect failures in individual units, as well as failures caused by defective manufacturing processes. Random defects in individual units may not significantly impact a company's balance sheet, but a defective manufacturing process for a complex circuit, or a design error in some obscure function, could escape detection until well after first customer shipments, resulting in a very expensive product recall.

Public safety must also be taken into account. Digital logic devices have become pervasive in products that affect public safety, including applications such as transportation and human implants. These products must be thoroughly tested to ensure that they are designed and fabricated correctly. Where design and test shared tools in the past, there is a steadily growing divergence in their methodologies. Formal verification techniques are emerging, and they are of particular importance in applications involving public safety.

Each new generation of EDA tools makes it possible to design and fabricate chips of greater complexity at lower cost. As a result, testing consumes a greater percentage of total production cost. It requires more effort to create a test program and requires more stimuli to exercise the chip. The difficulty in creating test programs for new designs also contributes to delays in getting products to the marketplace. Product managers must balance the consequences of delaying shipment of a product for which adequate test programs have not yet been developed against the consequences of shipping product and facing the prospect of wholesale failure and return of large quantities of defective products.

New test strategies are emerging in response to test problems arising from these increasingly complex devices, and greater emphasis is placed on finding defects as early as possible in the manufacturing cycle. New algorithms are being devised to create tests for logic circuits, and more attention is being given to design-for-test (DFT) techniques that require participation by logic designers, who are being asked to adhere to design rules that facilitate design of more testable circuits.

Built-in self-test (BIST) is a logical extension of DFT. It embeds test mechanisms directly into the product being designed, often using DFT structures. The goal is to place stimulus generation and response evaluation circuits closer to the logic being tested.

Fault tolerance also modifies the design, but the goal is to contain the effects of faults. It is used when it is critical that a product operate correctly. The goal of passive fault tolerance is to permit continued correct circuit operation in the presence of defects. Performance monitoring is another form of fault tolerance, sometimes called active fault tolerance, in which performance is evaluated by means of special self-testing circuits or by injecting test data directly into a device during operation. Errors in operation can be recognized, but recovery requires intervention by the processor or by an operator. An instruction may be retried or a unit removed from operation until it is repaired.

Remote diagnostics are yet another strategy employed in the quest for reliable computing. Some manufacturers of personal computers provide built-in diagnostics. If problems occur during operation and if the problem does not interfere with the ability to communicate via the modem, then the computer can dial a remote computer that is capable of analyzing and diagnosing the cause of the problem.

It should be obvious from the preceding paragraphs that there is no single solution to the test problem. There are many solutions, and a solution may be appropriate for one application but not for another. Furthermore, the best solution for a particular application may be a combination of available solutions. This requires that designers and test engineers understand the strengths and weaknesses of the various approaches.

THE ROADMAP

This textbook contains 12 chapters. The first six chapters can be viewed as building blocks. Topics covered include simulation, fault simulation, combinational and sequential test pattern generation, and a brief introduction to tester architectures. The last six chapters build on the first six. They cover design-for-test (DFT), built-in self-test (BIST), fault tolerance, memory test, I_{DDQ} test, and, finally, behavioral test and verification. This dichotomy represents a natural partition for a two-semester course. Some examples make use of the Verilog hardware design language (HDL). For those readers who do not have access to a commercial Verilog product, a quite good (and free) Verilog compiler/simulator can be downloaded from http://www.icarus.com. Every effort was made to avoid relying on advanced HDL concepts, so that the student familiar only with programming languages, such as C, can follow the Verilog examples.

PART I

Chapter 1 begins with some general observations about design, test, and quality. Acceptable quality level (AQL) depends both on the yield of the manufacturing processes and on the thoroughness of the test programs that are used to identify defective product. Process yield and test thoroughness are focal points for companies trying to balance quality, product cost, and time to market in order to remain profitable in a highly competitive industry.

Simulation is examined from various perspectives in Chapter 2. Simulators used in digital circuit design, like compilers for high-level languages, can be compiled or interpreted, with each having its distinct advantages and disadvantages. We start by looking at contemporary hardware design languages (HDL). Ironically, while software for personal computers has migrated from text to graphical interfaces, the input medium for digital circuits has migrated from graphics (schematic editors) to text. Topics include event-driven simulation and selective trace. Delay models for simulation include 0-delay, unit delay, and nominal delay. Switch-level simulation

represents one end of the simulation spectrum. Behavioral simulation and cycle simulation represent the other end. Binary decision diagrams (BDDs), used in support of cycle simulation, are introduced in this chapter. Timing analysis in synchronous designs is also discussed.

Chapter 3 concentrates on fault simulation algorithms, including parallel, deductive, and concurrent fault simulation. The chapter begins with a discussion of fault modeling, including, of course, the stuck-at fault model. The basic algorithms are examined, with a look at ways in which excess computations can be squeezed out of the algorithms in order to improve performance. The relationship between algorithms and the design environment is also examined: For example, how are the different algorithms affected by the choice of synchronous or asynchronous design environment?

The topic for Chapter 4 is automatic test pattern generation (ATPG) for combinational circuits. Topological, or path tracing, methods, including the D-algorithm with its formal notation, along with PODEM, FAN, and the critical path, are examined. The subscripted D-algorithm is examined; it represents an example of symbolic propagation. Algebraic methods are described next; these include Boolean difference and Boolean satisfiability. Finally, the use of BDDs for ATPG is discussed.

Sequential ATPG merits a chapter of its own. The search for an effective sequential ATPG has continued unabated for over a quarter-century. The problem is complicated by the presence of memory, races, and hazards. Chapter 5 focuses on some of the methods that have evolved to deal with sequential circuits, including the iterative test generator (ITG), the 9-value ITG, and the extended backtrace (EBT). We also look at some experiments on state machines, including homing sequences, distinguishing sequences, and so on, and see how these lead to circuits which, although testable, require more information than is available from the netlist.

Chapter 6 focuses on automatic test equipment. Testers in use today are extraordinarily complex; they have to be in order to keep up with the ICs and PCBs in production; hence this chapter can be little more than a brief overview of the subject. Testers are used to test circuits in production environments, but they are also used to characterize ICs and PCBs. In order to perform characterization, the tester must be able to operate fast enough to clock the circuit at its intended speed, it must be able to accurately measure current and voltage, and it must be possible to switch input levels and strobe output pins in a matter of picoseconds. The Standard Test Interface Language (STIL) is also examined in this chapter. Its goal it to give a uniform appearance to the many different tester architectures on the marketplace.

PART II

Topics covered in the first six chapters, including logic and fault simulators, ATPG algorithms, and the various testers and test strategies, can be thought of as building blocks, or components, of a successful test strategy. In Chapter 7 we bring these components together in order to determine how to leverage the tools, individually

and in conjunction with other tools, in order to create a successful test strategy. This often requires an understanding of the environment in which they function, including such things as design methodologies, HDLs, circuit models, data structures, and fault modeling strategies. Different technologies and methodologies require very different tools.

The focus up to this point has been on the traditional approach to test—that is, apply stimuli and measure response at the output pins. Unfortunately, existing algorithms, despite decades of research, remain ineffective for general sequential logic. If the algorithms cannot be made powerful enough to test sequential logic, then circuit complexity must be reduced in order to make it testable. Chapters 8 and 9 look at ways to improve testability by altering the design in order to improve access to its inner workings. The objectives are to make it easier to apply a test (improve controllability) and make it easier to observe test results (improve observability). Design-for-test (DFT) makes it easier to develop and apply tests via conventional testers. Built-in self-test (BIST) attempts to replace the tester, or at least offload many of its tasks. Both methodologies make testing easier by reducing the amount and/or complexity of logic through which a test must travel either to stimulate the logic being tested or to reach an observable output whereby the test can be monitored.

Memory test is covered in Chapter 10. These structures have their own problems and solutions as a result of their regular, repetitive structure and we examine some algorithms designed to exploit this regularity. Because memories keep growing in size, the memory test problem continues to escalate. The problem is further exacerbated by the fact that increasingly larger memories are being embedded in microprocessors and other devices. In fact, it has been suggested that as microprocessors grow in transistor count, they are becoming de facto memories with a little logic wrapped around them. A growing trend in memories is the use of memory BIST (MBIST). This chapter contains two Verilog implementations of memory test algorithms.

Complementary metal oxide semiconductor (CMOS) circuits draw little or no current except when clocked. Consequently, excessive current observed when an IC is in the quiescent state is indicative of either a hard failure or a potential reliability problem. A growing number of investigators have researched the implications of this observation, and determined how to leverage this potentially powerful test strategy. I_{DDQ} will be the focus of Chapter 11.

Design verification and test can be viewed as complementary aspects of one problem, namely, the delivery of reliable computation, control, and communications in a timely and cost-effective manner. However, it is not completely obvious how these two disciplines are related. In Chapter 12 we look closely at design verification. The opportunities to leverage test development methodologies and tools in design verification—and, conversely, the opportunities to leverage design verification efforts to obtain better test programs—make it essential to understand the relationships between these two efforts. We will look at some evolving methodologies and some that are maturing, and we will cover some approaches best described as ongoing research.

The goal of this textbook is to cover a representative sample of algorithms and practices used in the IC industry to identify faulty product and prevent, to the extent possible, *tester escapes*—that is, faulty devices that slip through the test process and make their way into the hands of customers. However, digital test is not a "one size fits all" industry.

Given two companies with similar digital products, test practices may be as different as day and night, and yet both companies may have rational test plans. Minor nuances in product manufacturing practices can dictate very different strategies. Choices must be made everywhere in the design and test cycle. Different individuals within the same project may be using simulators ranging from switch-level to cycle-based. Testability enhancements may range from ad hoc techniques, to partial-scan, to full-scan. Choices will be dictated by economics, the capabilities of the available tools, the skills of the design team, and other circumstances.

One of the frustrations faced over the years by those responsible for product quality has been the reluctance on the part of product planners to face up to and address test issues. Nearly 500 years ago Nicolo Machiavelli, in his book *The Prince*, observed that "fevers, as doctors say, at their beginning are easy to cure but difficult to recognise, but in course of time when they have not at first been recognised, and treated, become easy to recognise and difficult to cure.[5]" In a similar vein, in the early stages of a design, test problems are difficult to recognize but easy to solve; further into the process, test problems become easier to recognize but more difficult to cure.

REFERENCES

1. Brandt, R., The Birth of Intel's Pentium Chip—and the Labor Pains, *Business Week*, March 29, 1993, pp. 94–95.
2. Bass, Michael J., and Clayton M. Christensen, The Future of the Microprocessor Business, *IEEE Spectrum*, Vol. 39, No. 4, April 2002, pp. 34–39.
3. Port, O., Gordon Moore's Crystal Ball, *Business Week*, June 23, 1997, p. 120.
4. Foderaro, J. K., K. S. Van Dyke, and D. A. Patterson, Running RISCs, *VLSI Des.*, September–October 1982, pp. 27–32.
5. Machiavelli, Nicolo, The Prince and the Discourses, in *The Prince*, Chapter 3, Random House, 1950.

CHAPTER 1

Introduction

1.1 INTRODUCTION

Things don't always work as intended. Some devices are manufactured incorrectly, others break or wear out after extensive use. In order to determine if a device was manufactured correctly, or if it continues to function as intended, it must be tested. The test is an evaluation based on a set of requirements. Depending on the complexity of the product, the test may be a mere perusal of the product to determine whether it suits one's personal whims, or it could be a long, exhaustive checkout of a complex system to ensure compliance with many performance and safety criteria. Emphasis may be on speed of performance, accuracy, or reliability.

Consider the automobile. One purchaser may be concerned simply with color and styling, another may be concerned with how fast the automobile accelerates, yet another may be concerned solely with reliability records. The automobile manufacturer must be concerned with two kinds of test. First, the design itself must be tested for factors such as performance, reliability, and serviceability. Second, individual units must be tested to ensure that they comply with design specifications.

Testing will be considered within the context of digital logic. The focus will be on technical issues, but it is important not to lose sight of the economic aspects of the problem. Both the cost of developing tests and the cost of applying tests to individual units will be considered. In some cases it becomes necessary to make trade-offs. For example, some algorithms for testing memories are easy to create; a computer program to generate test vectors can be written in less than 12 hours. However, the set of test vectors thus created may require several millenia to apply to an actual device. Such a test is of no practical value. It becomes necessary to invest more effort into initially creating a test in order to reduce the cost of applying it to individual units.

This chapter begins with a discussion of quality. Once we reach an agreement on the meaning of quality, as it relates to digital products, we shift our attention to the subject of testing. The test will first be defined in a broad, generic sense. Then we put the subject of digital logic testing into perspective by briefly examining the overall design process. Problems related to the testing of digital components and

Digital Logic Testing and Simulation, *Second Edition*, by Alexander Miczo
ISBN 0-471-43995-9 Copyright © 2003 John Wiley & Sons, Inc.

assemblies can be better appreciated when viewed within the context of the overall design process. Within this process we note design stages where testing is required. We then look at design aids that have evolved over the years for designing and testing digital devices. Finally, we examine the economics of testing.

1.2 QUALITY

Quality frequently surfaces as a topic for discussion in trade journals and periodicals. However, it is seldom defined. Rather, it is assumed that the target audience understands the intended meaning in some intuitive way. Unfortunately, intuition can lead to ambiguity or confusion. Consider the previously mentioned automobile. For a prospective buyer it may be deemed to possess quality simply because it has a soft leather interior and an attractive appearance. This concept of quality is clearly subjective: It is based on individual expectations. But expectations are fickle: They may change over time, sometimes going up, sometimes going down. Furthermore, two customers may have entirely different expectations; hence this notion of quality does not form the basis for a rigorous definition.

In order to measure quality quantitatively, a more objective definition is needed. We choose to define quality as the degree to which a product meets its requirements. More precisely, it is the degree to which a device conforms to applicable specifications and workmanship standards.[1] In an integrated circuit (IC) manufacturing environment, such as a wafer fab area, quality is the absence of "drift"—that is, the absence of deviation from product specifications in the production process. For digital devices the following equation, which will be examined in more detail in a later section, is frequently used to quantify quality level:[2]

$$\text{AQL} = Y^{(1-T)} \tag{1.1}$$

In this equation, AQL denotes acceptable quality level, it is a function of Y (product yield) and T (test thoroughness). If no testing is done, AQL is simply the *yield*—that is, the number of good devices divided by the total number of devices made. Conversely, if a complete test were created, then $T = 1$, and all defects are detected so no bad devices are shipped to the customer.

Equation (1.1) tells us that high quality can be realized by improving product yield and/or the thoroughness of the test. In fact, if $Y \geq \text{AQL}$, testing is not required. That is rarely the case, however. In the IC industry a high yield is often an indication that the process is not aggressive enough. It may be more economically rewarding to shrink the geometry, produce more devices, and screen out the defective devices through testing.

1.3 THE TEST

In its most general sense, a test can be viewed as an experiment whose purpose is to confirm or refute a hypothesis or to distinguish between two or more hypotheses.

Figure 1.1 depicts a test configuration in which stimuli are applied to a device-under-test (DUT), and the response is evaluated. If we know what the *expected response* is from the correctly operating device, we can compare it to the response of the DUT to determine if the DUT is responding correctly.

When the DUT is a digital logic device, the stimuli are called *test patterns* or *test vectors*. In this context a *vector* is an ordered *n*-tuple; each bit of the vector is applied to a specific input pin of the DUT. The expected or predicted outcome is usually observed at output pins of the device, although some test configurations permit monitoring of test points within the circuit that are not normally accessible during operation. A tester captures the response at the output pins and compares that response to the expected response determined by applying the stimuli to a known good device and recording the response, or by creating a *model* of the circuit (i.e., a representation or abstraction of selected features of the system[3]) and simulating the input stimuli by means of that model. If the DUT response differs from the expected response, then an *error* is said to have occurred. The error results from a *defect* in the circuit.

The next step in the process depends on the type of test that is to be applied. A taxonomy of test types[4] is shown in Table 1.1. The classifications range from testing die on a bare wafer to tests developed by the designer to verify that the design is correct. In a typical manufacturing environment, where tests are applied to die on a wafer, the most likely response to a failure indication is to halt the test immediately and discard the failing part. This is commonly referred to as a go–nogo test. The object is to identify failing parts as quickly as possible in order to reduce the amount of time spent on the tester.

If several functional test programs were developed for the part, a common practice is to arrange them so that the most effective test program—that is, the one that uncovers the most defective parts—is run first. Ranking the effectiveness of the test programs can be done through the use of a fault simulator, as will be explained in a subsequent chapter. The die that pass the wafer test are packaged and then retested. Bonding a chip to a package has the potential to introduce additional defects into the process, and these must be identified.

Binning is the practice of classifying chips according to the fastest speed at which they can operate. Some chips, such as microprocessors, are priced according to their clock speed. A chip with a 10% performance advantage may bring a 20–50% premium in the marketplace. As a result, chips are likely to first be tested at their maximum rated speed. Those that fail are retested at lower clock speeds until either they pass the test or it is determined that they are truly defective. It is, of course, possible that a chip may run successfully at a clock speed lower than any for which it was tested. However, such chips can be presumed to have no market value.

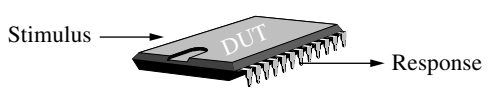


Figure 1.1 Typical test configuration.

TABLE 1.1 Types of Tests

Type of Test	Purpose of Test
Production	Test of manufactured parts to sort out those that are faulty
Wafer Sort or Probe	Test of each die on the wafer.
Final or Package	Test of packaged chips and separation into bins (military, commercial, industrial).
Acceptance	Test to demonstrate the degree of compliance of a device with purchaser's requirements.
Sample	Test of some but not all parts.
Go–nogo	Test to determine whether device meets specifications.
Characterization or engineering	Test to determine actual values of AC and DC parameters and the interaction of parameters. Used to set final specifications and to identify areas to improve process to increase yield.
Stress screening (burn-in)	Test with stress (high temperature, temperature cycling, vibration, etc.) applied to eliminate short life parts.
Reliability (accelerated life)	Test after subjecting the part to extended high temperature to estimate time to failure in normal operation.
Diagnostic (repair)	Test to locate failure site on failed part.
Quality	Test by quality assurance department of a sample of each lot of manufactured parts. More stringent than final test.
On-line or checking	On-line testing to detect errors during system operation.
Design verification	Verify the correctness of a design.

Diagnosis may be called for when there is a yield crash—that is, a sudden, significant drop in the number of devices that pass a test. To aid in investigating the causes, it may be necessary to create additional test vectors specifically for the purpose of isolating the source of the crash. For ICs it may be necessary to resort to an e-beam probe to identify the source. Production diagnostic tests are more likely to be created for a printed circuit board (PCB), since they are often repairable and generally represent a larger manufacturing cost. Tests for memory arrays are thorough and methodical, thus serving both as go–no-go tests and as diagnostic tests. These tests permit substitution of spare rows or columns in order to repair the memory array, thereby significantly improving the yield.

Products tend to be more susceptible to yield problems in the early stages of their existence, since manufacturing processes are new and unfamiliar to employees. As a result, there are likely to be more occasions when it is necessary to investigate problems in order to diagnose causes. For mature products, yield is frequently quite high, and testing may consist of sampling by randomly selecting parts for test. This is also a reasonable strategy for low complexity parts, such as a chip that goes into a wristwatch.

To protect against yield problems, particularly in the early phases of a project, *burn-in* is commonly employed. Burn-in stresses semiconductor products in order to

identify and eliminate marginal performers. The goal is to ensure the shipment of parts having an acceptably low failure rate and to potentially improve product reliability.[5] Products are operated at environmental extremes, with the duration of this operation determined by product history. Manufacturers institute programs, such as Intel's ZOBI (zero hour burn-in), for the purpose of eliminating burn-in and the resulting capital equipment costs.[6]

When stimuli are simulated against the circuit model, the simulator produces a file that contains the input stimuli and expected response. This information goes to the tester, where the stimuli are applied to manufactured parts. However, this information does not provide any indication of just how effective the test is at detecting defects internal to the circuit. Furthermore, if an erroneous response should occur at any of the output pins during testing of manufactured parts, there is no insight into the location of the defect that induced the incorrect response. Further testing may be necessary to distinguish which of several possible defects produced the response. This is accomplished through the use of fault models.

The process is essentially the same; that is, vectors are simulated against a model of the circuit, except that the computer model is modified to make it appear as though a fault were present. By simulating the correct model and the faulted model, responses from the two models can be compared. Furthermore, by injecting several faults into the model, one at a time, and then simulating, it is possible to compare the response of the DUT to that of the various faulted models in order to determine which faulted model either duplicates or most closely approximates the behavior of the DUT.

If the DUT responds correctly to all applied stimuli, confidence in the DUT increases. However, we cannot conclude that the device is fault-free! We can only conclude that it does not contain any of the faults for which it was tested, but it could contain other faults for which an effective test was not applied.

From the preceding paragraphs it can be seen that there are three major aspects of the test problem:

1. Specification of test stimuli
2. Determination of correct response
3. Evaluation of the effectiveness of the stimuli

Furthermore, this approach to testing can be used both to detect the presence of faults and to distinguish between several faults for repair purposes.

In digital logic, the three phases of the test process listed above are referred to as test pattern generation, logic simulation, and fault simulation. More will be said about these processes in later chapters. For the moment it is sufficient to state that each of these phases ranks equally in importance; they in fact complement one another. Stimuli capable of distinguishing between good circuits and faulted circuits do not become effective until they are simulated so their effects can be determined. Conversely, extremely accurate simulation against very precise models with

ineffective stimuli will not uncover many defects. Hence, measuring the effectiveness of test stimuli, using an accepted metric, is another very important task.

1.4 THE DESIGN PROCESS

Table 1.1 identifies several types of tests, ranging from design verification, whose purpose is to ensure that a design conforms to the designer's intent, to various kinds of tests directed toward identifying units with manufacturing defects, and tests whose purpose is to identify units that develop defects during normal usage. The goal during product design is to develop comprehensive test programs before a design is released to manufacturing. In reality, test programs are not always adequate and may have to be enhanced due to an excessive number of faulty units reaching end users. In order to put test issues into proper perspective, it will be helpful here to take a brief look at the design process, starting with initial product conception.

A digital device begins life as a concept whose eventual goal is to fill a perceived need. The concept may flow from an original idea or it may be the result of market research aimed at obtaining suggestions for enhancements to an existing product. Four distinct product development classifications have been identified:[7]

First of a kind
Me too with a twist
Derivative
Next-generation product

The "first of a kind" is a product that breaks new ground. Considerable innovation is required before it is implemented. The "me too with a twist" product adds incremental improvements to an existing product, perhaps a faster bus speed or a wider data path. The "derivative" is a product that is derived from an existing product. An example would be a product that adds functionality such as video graphics to a core microprocessor. Finally, the "next-generation product" replaces a mature product. A 64-bit microprocessor may subsume op-codes and basic capabilities, but also substantially improve on the performance and capabilities of its 32-bit predecessor.

The category in which a product falls will have a major influence on the design process employed to bring it to market. A "first of a kind" product may require an extensive requirements analysis. This results in a detailed product specification describing the functionality of the product. The object is to maximize the likelihood that the final product will meet performance and functionality requirements at an acceptable price. Then, the behavioral description is prepared. It describes what the product will do. It may be brief, or it may be quite voluminous. For a complex design, the product specification can be expected to be very formal and detailed. Conversely, for a product that is an enhancement to an existing product, documentation may consist of an engineering change notice describing only the proposed changes.

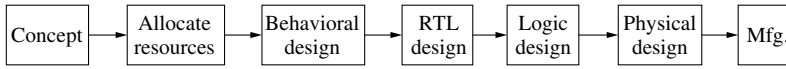


Figure 1.2 Design flow.

After a product has been defined and a decision has been made to manufacture and market the device, a number of activities must occur, as illustrated in Figure 1.2. These activities are shown as occurring sequentially, but frequently the activities overlap because, once a commitment to manufacture has been made, the objective is to get the product out the door and into the marketplace as quickly as possible. Obviously, nothing happens until a development team is put in place. Sometimes the largest single factor influencing the time-to-market is the time required to allocate resources, including staff to implement the project and the necessary tools by which the staff can complete the design and put a manufacturing flow into place. For a device with a given level of performance, time of delivery will frequently determine if the product is competitive; that is, does it fall above or below the performance–time plot illustrated in Figure 1.3?

Once the behavioral specification has been completed, a functional design must be created. This is actually a continuous flow; that is, the behavior is identified, and then, based on available technology, architects identify functional units. At that stage of development an important decision must be made as to whether or not the product can meet the stated performance objectives, given the architecture and technology to be used. If not, alternatives must be examined. During this phase the logic is partitioned into physical units and assigned to specific units such as chips, boards, or cabinets. The partitioning process attempts to minimize I/O pins and cabling between chips, boards, and units. Partitioning may also be used to advantage to simplify such things as test, component placement, and wire routing.

The use of hardware design languages (HDLs) for the design process has become virtually universal.Two popular HDLs, VHDL (VHSIC Hardware Description Language) and Verilog, are used to

Specify an architecture
Partition the architecture into smaller modules
Synthesize an RTL description
Verify that a structural implementation corresponds to the architectural design
Check out microcode and/or diagnostic programs
Serve as documentation

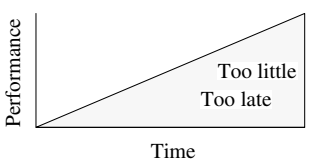


Figure 1.3 Performance–time plot.

A behavioral description specifies what a design must do. There is usually little or no indication as to how it must be done. For example, a large case statement might identify operations to be performed by an ALU in response to different values applied to a control field. The RTL design refines the behavioral description. Operations identified at the behavioral level are elaborated upon in more detail. RTL design is followed by logic design. This stage may be generated by synthesis programs, or it may be created manually, or, more often, some modules are synthesized while others are manually designed or included from a library of predesigned modules, some or all of which may have been purchased from an outside vendor. The use of predesigned, or core, modules may require selecting and/or altering components and specifying the interconnection of these components. At the end of the process, it may be the case that the design will not fit on a piece of silicon, or there may not be enough I/O pins to accommodate the signals, in which case it becomes necessary to reevaluate the design.

Physical design specifies the physical placement of components and the routing of wires between components. Placement may assign circuits to specific areas on a piece of silicon, it may specify the placement of chips on a PCB, or it may specify the assignment of PCBs to a cabinet. The routing task specifies the physical connection of devices after they have been placed. In some applications, only one or two connection layers are permitted. Other applications may permit PCBs with 20 or more interconnection layers, with alternating layers of metal interconnects and insulating material.

The final design is sent to manufacturing, where it is fabricated. Engineering changes must frequently be accommodated due to logic errors or other unexpected problems such as noise, timing, heat buildup, electrical interference, and so on, or inability to mass produce some critical parts.

In these various design stages there is a continuing need for testing. Requirements analysis attempts to determine whether the product will fulfill its objectives, and testing techniques are frequently based on marketing studies. Early attempts to introduce more rigor into this phase included the use of design languages such as PSL/PSA (Problem Statement Language/Problem Statement Analyzer).[8] It provided a way both to rigorously state the problem and to analyze the resulting design. PMS (Processors, Memories, Switches)[9] was another early attempt to introduce rigor into the initial stages of a design project, permitting specification of a design via a set of consistent and systematic rules. It was often used to evaluate architectures at the system level, measuring data throughput and looking for design bottlenecks. Verilog and VHDL have become the standards for expressing designs at all levels of abstraction, although investigation into specification languages continues to be an active area of research. Its importance is seen from such statements as "requirements errors typically comprise over 40% of all errors in a software project"[10] and "the really serious mistakes occur in the first day."[3]

A design expressed in an HDL, at a level of abstraction that describes intended behaviors, can be formally tested. At this level the design is a requirements document that states, in a simulation language, what actions the product must perform. The HDL permits the designer to simulate behavioral expressions with input vectors

chosen to confirm correctness of the design or to expose design errors. The design verification vectors must be sufficient to confirm that the design satisfies the behavior expressed in the product specification. Development of effective test stimuli at this state is highly iterative; a discrepancy between designer intent and simulation results often indicates the need for more stimuli to diagnose the underlying reason for the discrepancy. A growing trend at this level is the use of formal verification techniques (cf. Chapter 12.)

The logic design is tested in a manner similar to the functional design. A major difference is that the circuit description is more detailed; hence thorough analysis requires that simulations be more exhaustive. At the logic level, timing is of greater concern, and stimuli that were effective at the register transfer level (RTL) may not be effective in ferreting out critical timing problems. On the other hand, stimuli that produced correct or expected response from the RTL circuit may, when simulated by a timing simulator, indicate incorrect response or may indicate marginal performance, or the simulator may simply indicate that it cannot predict the correct response.

The testing of physical structure is probably the most formal test level. The test engineer works from a detailed design document to create tests that determine if response of the fabricated device corresponds to response of the design. Studies of fault behavior of the selected circuit family or technology permit the creation of fault models. These fault models are then used to create specific test stimuli that attempt to distinguish between the correctly operating device and a device with the fault.

This last category, which is the most highly developed of the design stages, due to its more formal and well-defined environment, is where we will concentrate our attention. However, many of the techniques that have been developed for structural testing can be applied to design verification at the logic and functional levels.

1.5 DESIGN AUTOMATION

Many of the activities performed by architects and logic designers were long ago recognized to be tedious, repetitious, error prone, and time-consuming, and hence could and should be automated. The mechanization of tedious design processes reduces the potential for errors caused by human fatigue, boredom, and inattention to mundane details. Early elimination of errors, which once was a desirable objective, has now become a virtual necessity. The market window for new products is sometimes so small that much of that window will have evaporated in the time that it takes to correct an error and push the design through the entire fabrication cycle yet another time.

In addition to the reduction of errors, elimination of tedious and time-consuming tasks enables designers to spend more time on creative endeavors. The designer can experiment with different solutions to a problem before a design becomes frozen in silicon. Various alternatives and trade-offs can be studied. This process of automating various aspects of the design process has come to be known as *electronic design*

*automatio*n (EDA). It does not replace the designer but, rather, enables the designer to be more productive and more creative. In addition, it provides access to IC design for many logic designers who know very little about the intricacies of laying out an IC design. It is one of the major factors responsible for taking cost out of digital products.

Depending on whether it is an IC, a PCB, or a system comprised of several PCBs, a typical EDA system supports some or all of the following capabilities:

- Data management
  - Record data
  - Retrieve data
  - Define relationships
  - Perform rules checks
- Design analysis/verification
  - Evaluate performance/capabilities
  - Simulate
  - Check timing
- Design fabrication
  - Perform placement and routing
  - Create tests for structural defects
  - Identify qualified vendors
- Documentation
  - Extract parts list
  - Create/update product specification

The data management system supports a data base that serves as a central repository for all design data. A data management program accepts data from the designer, formats it, and stores it in the data base. Some validity checks can be performed at this time to spot obvious errors. Programs must be able to retrieve specific records from the data base. Different applications require different records or combinations or records. As an example, one that we will elaborate on in a later chapter, a test program needs information concerning the specific ICs used in the design of a board, it needs information concerning their interconnections, and it needs information concerning their physical location on a board.

A data base should be able to express hierarchical relationships.[11] This is especially true if a facility designs and fabricates both boards and ICs. The ICs are described in terms of logic gates and their interconnections, while the board is described in terms of ICs and their interconnections. A "where used" capability for a part number is useful if a vendor provides notice that a particular part is no longer available. Rules checks can include examination of fan-out from a logic gate to ensure that it does not exceed some specified limit. The total resistive or capacitive loading on an output can be checked. Wire length may also be critical in some applications, and rules checking programs should be able to spot nets that exceed wire length maximums.

The data management system must be able to handle multiple revisions of a design or multiple physical implementations of a single architecture. This is true for manufacturers who build a range of machines all of which implement the same architecture. It may not be necessary to maintain an architectural level copy with each physical implementation. The system must be able to control access and update to a design, both to protect proprietary design information from unauthorized disclosure and to protect the data base from inadvertent damage. A lock-out mechanism is useful to prevent simultaneous updates that could result in one or both of the updates being lost.

Design analysis and verification includes simulation of a design after it is recorded in the data base to verify that it is functionally correct. This may include RTL simulation using a hardware design language and/or simulation at a gate level with a logic simulator. Precise relationships must be satisfied between clock and data paths. After a logic board with many components is built, it is usually still possible to alter the timing of critical paths by inserting delays on the board. On an IC there is no recourse but to redesign the chip. This evaluation of timing can be accomplished by simulating input vectors with a timing simulator, or it can be done by tracing specific paths and summing up the delays of elements along the way.

After a design has stabilized and has been entered into a data base, it can be fabricated. This involves placement either of chips on a board or of circuits on a die and then interconnecting them. This is usually accomplished by placement and routing programs. The process can be fully automated for simple devices, or for complex devices it may require an interactive process whereby computer programs do most of the task, but require the assistance of an engineer to complete the task. Checking programs are used after placement and routing.

Typical checks look for things such as runs too close to one another, and possible opens or shorts between runs. After placement and routing, other kinds of analysis can be performed. This includes such things as computing heat concentration on an IC or PCB and computing the reliability of an assembly based on the reliability of individual components and manufacturing processes. Testing the structure involves creation of test stimuli that can be applied to the manufactured IC or PCB to determine if it has been fabricated correctly.

Documentation includes the extraction of parts lists, the creation of logic diagrams and printing of RTL code. The parts list is used to maintain an inventory of parts in order to fabricate assemblies. The parts list may be compared against a master list that includes information such as preferred vendors, second sources, or alternate parts which may be used if the original part is unavailable. Preferred vendors may be selected based on an evaluation of their timeliness in delivering parts and the quality of parts received from them in the past. Logic diagrams are used by technicians and field engineers to debug faulty circuits as well as by the original designer or another designer who must modify or debug a logic design at some future date.

1.6 ESTIMATING YIELD

We now look at yield analysis, based on various probability distribution functions. But, first, just how important are yield equations? James Cunningham[12] describes a

situation in which a company was invited to submit a bid to manufacture a large CMOS custom logic chip. The chip had already been designed at another company and was to have a die area of 2.3 cm^2. The company had experience making CMOS parts, but never one this large. Hence, they were uncertain as to how to estimate yield for a chip of this size.

When they extrapolated from existing data, using a computer-generated best-fit model, they obtained a yield estimate $Y = 1.4\%$. Using a Poisson model with $D_0 = 2.1$, where D_0 is the average number of defects per unit area A, they obtained an estimate $Y = 0.8\%$. They then calculated the yield using Seeds' model,[13] which gave $Y = 17\%$. That was followed by Murphy's model.[14] It gave $Y = 4\%$. They decided to average Seeds' model and Murphy's model and submit a bid based on 11% die sort yield. A year later they were producing chips with a yield of 6%, even though D_0 had fallen from 2.1 to 1.9 defects/cm^2. The company had started to evaluate the negative binomial yield model $Y = (1 + D_0A/\alpha)^{-\alpha}$. A value of $\alpha = 3$ produced a good fit for their yield data. Unfortunately, the company could not sustain losses on the product and dropped it from production, leaving the customer without a supply of parts.

Probability distribution functions are used to estimate the probability of an event occurring. The *binomial probability distribution* is a discrete distribution, which is expressed as

$$P(k) = \frac{n!}{k!(n-k)!}P^k(1-P)^{n-k} \tag{1.2}$$

If P is the probability of a defect on a die, then $P(k)$ is the probability of k defects on the die, when there are a total of $n = D_0A_w$ defects, where A_w is the area of the wafer. The probability P is $D_0A/D_0A_w = A/A_w$. Substituting into Eq. (1.2) yields

$$P(k) = \frac{n!}{k!(n-k)!}\left(\frac{A}{A_w}\right)^k\left(1-\frac{A}{A_w}\right)^{n-k} \tag{1.3}$$

To derive the equation for a die with no defects, set $k = 0$. This yields

$$P(k=0) = \left[1-\frac{A}{A_w}\right]^{D_0A_w} \tag{1.4}$$

The first distribution that was frequently used to estimate yields was the *Poisson distribution*, which is expressed as

$$P(k) = \frac{e^{-\lambda_0}\lambda_0^{\,k}}{k!} \qquad \text{for } k = 0, 1, 2, \ldots \tag{1.5}$$

where λ_0 is the average number of defects per die. For die with no defects ($k = 0$), the equation becomes $P(0) = e^{-\lambda_0}$. If $\lambda_0 = .5$, the yield is predicted to be .607. In general, the Poisson distribution requires that defects be uniformly and randomly distributed. Hence, it tends to be pessimistic for larger die sizes. Considering again

the binomial distribution, if the number of trials, n, is large, and the probability p of occurrence of an event is close to zero, then the binomial distribution is closely approximated by the Poisson distribution with $\lambda = n \cdot p$.

Another distribution commonly used to estimate yield is the *normal distribution*, also known as the *Gaussian distribution*. It is the familiar bell-shaped curve and is expressed as

$$P(k) = \frac{1}{\sigma\sqrt{2\pi}} e^{-(k-\mu)^2/2\sigma^2} \qquad (-\infty < k < \infty) \tag{1.6}$$

The variable μ represents the mean, σ represents the standard deviation, and σ^2 represents the variance. If n is large and if neither p or q is too close to zero, the binomial distribution can be closely approximated by a normal distribution. This can be expressed as

$$\lim_{n \to \infty} P\left(a \le \frac{x - np}{\sqrt{npq}} \le b\right) = \frac{1}{\sqrt{2\pi}} \int_a^b e^{-u^2/2} du \tag{1.7}$$

where np represents the mean for the binomial distribution, $\sqrt{npq}$ is the standard deviation, npq is the variance, and x is the number of successful trials.

When Murphy investigated the yield problem in 1964, he observed that defect and particle densities vary widely among chips, wafers, and runs. Under these circumstances, the Poisson model is likely to underestimate yield, so he chose to use the normalized probability distribution function. To derive a yield equation, Murphy multiplied the probability distribution function with the probability p that the device was good, for a given defect density D, and then summed that over all values of D, that is,

$$Y = \int_0^\infty pf(D) dD \tag{1.8}$$

He substituted $p = e^{-Da}$ for the probability that the device was good. However, he could not integrate the bell-shaped curve, so he approximated it with a triangle function. This gave

$$Y = \left(\frac{1 - e^{-D_0 A}}{D_0 A}\right)^2 \tag{1.9}$$

By substituting other expressions for $f(D)$ in Eq. (1.8), other yield equations result. Seeds used an exponential distribution function $f(D) = e^{-D/D_0}/D_0$. Substituting this into Eq. (1.8), he obtained

$$Y = \frac{1}{1 + D_0 A} \tag{1.10}$$

In 1973 Charles Stapper[15] derived a yield equation that is often referred to as a negative binomial distribution. By substituting $p(x) = e^{-\lambda}\lambda^x/x!$ and the gamma

distribution function $f(\lambda) = \frac{1}{\Gamma(\alpha)\beta^{\alpha}}\lambda^{\alpha-1}e^{-\lambda/\beta}$ into Murphy's equation [Eq. (1.8)] and integrating, he obtained

$$Y = (1 + D_0 A/\alpha)^{-\alpha} \tag{1.11}$$

The mean of the gamma function is given by $\mu = \alpha/\lambda$, whereas the variance is given by α/λ^2. Compare these with the mean and variance of the negative binomial distribution, sometimes referred to as Pascal's distribution: mean = nq/p and variance = nq/p^2.

The parameter α in Eq. (1.11) is referred to as the cluster parameter. By selecting appropriate values of α, the other yield equations can be approximated by Eq. (1.11). The value of α can be determined through statistical analysis of defect distribution data, permitting an accurate yield model to be obtained.

1.7 MEASURING TEST EFFECTIVENESS

In this chapter the intent has been to survey some of the many approaches to digital logic test. The objective is to illustrate how these approaches fit together to produce a program targeted toward product quality. Hence, we have touched only briefly on many topics that will be covered in greater detail in subsequent chapters. One of the topics examined here is fault modeling. It has been the practice, for over three decades, to resort to the use of stuck-at models to imitate the effects of defects. This model was more realistic when (small-scale integration) (SSI) was predominant. However, the stuck-at model, for practical reasons, is still widely used by commercial tools. Basically put, this model assumes that an input or output of a logic gate (e.g., an inverter, an AND gate, an OR gate, etc.) is stuck to a logic value 0 or 1 and is insensitive to signal changes from the signal that drives it.

With this faulting mechanism the process, in rather general terms, proceeds as follows: Computer models of digital circuits are created, and faults are injected into the model. The fault-free circuit and the faulted circuit are simulated. If there is a difference in response at an observable I/O pin, the fault is classified as detected. After many faults are evaluated in this manner, fault coverage is computed as

Fault coverage = No. faults detected / No.faults modeled

Given a fault coverage number, there are two questions that occur: How accurate is it, and for a given fault coverage, how many defective chips are likely to become tester escapes? Accuracy of fault coverage will depend on the faults selected and the accuracy of the fault model relative to real defect mechanisms. Fault selection requires a statistically meaningful random sample, although it is often the practice to

fault simulate a universal sample of faults, meaning faults applied to all logic elements in a circuit. The fault model, like any model, is an imperfect replica. It is rather simplistic when compared to the various, complex kinds of defects that can occur in a circuit; therefore, predictions of test effectiveness based on the stuck-at model are prone to error and imprecision. The number of tester escapes will depend on the thoroughness of the test—that is, the fault coverage, the accuracy of that fault coverage, and the process yield.

The term *defect level* (DL) is used to denote the fraction of shipped ICs that are bad. It is computed as

$$DL = \text{Number of faulty units shipped} / \text{Total no. units shipped} \tag{1.12}$$

It has also been variously referred to as *field reject rate* and *reject ratio*. In this section we adhere to the terminology used by the original authors in their derivations.

Over the past two decades a number of attempts have been made to quantify the effectiveness of test programs—that is, determine how many defective chips will be detected by the tester and how many will slip through the test process and reach the end user. Different researchers have come up with different equations for computing defect level. The discrepancies are based on the fact that they start with different assumptions about fault distributions. Some of it is a result of basing results on different technologies, and some of it is a result of working with processes that have different quality levels, different failure mechanisms, and/or different defect distributions. We present here a survey of some of the equations that have been derived over the years to compute defect level as a function of process yields and test coverage.

In 1978 Wadsack[16] derived the following equation:

$$yr = (1 - f) \cdot (1 - y) \tag{1.13}$$

where yr denotes the field reject rate—that is, the fraction of defective chips that passed the test and were shipped to the customer. The variable y, $0 \le y \le 1$, denotes the actual yield of the process, and f, $0 \le f \le 1$, denotes the fault coverage. In 1981 Williams and Brown developed the following equation:

$$DL = 1 - Y^{(1-T)} \tag{1.14}$$

In this equation the field reject rate is DL (defect level), the variable Y represents the yield of the manufacturing process, and the variable T represents the test percentage where, as in Eq. (1.13), each of these is a fraction between 0 and 1.

Example If it were possible to test for all defects, then

$$f = 1 \quad \text{and} \quad yr = (1 - 1) \cdot (1 - y) = 0 \qquad \text{from Eq. (1.13)}$$

$$T = 1 \quad \text{and} \quad DL = 1 - Y^{(1-1)} = 0 \qquad \text{from Eq. (1.14)}$$

On the other hand, if no defective units were manufactured, then

$$y = 1 \quad \text{and} \quad yr = (1 - f) \cdot (1 - 1) = 0 \qquad \text{from Eq. (1.13)}$$
$$Y = 1 \quad \text{and} \quad DL = 1 - 1^{(1-T)} = 0 \qquad \text{from Eq. (1.14)}$$

In either situation, no defective units are shipped, regardless of which equation is used. ■■

For either of these equations, if the yield is known, it is possible to find the fault coverage required to achieve a desired defect level. Using Eq. (1.14), the test fraction T is

$$T = 1 - \frac{\log(1 - DL)}{\log(Y)} \tag{1.15}$$

Example Integrated circuits (ICs) are manufactured on wafers—round, thin silicon substrates. After processing, individual ICs are tested. The wafer is diced and the die that tested bad are discarded. If the yield of good die is 60%, and we want a defect level not to exceed 0.1%, what level of testing must we achieve? Using Eq. (1.15), we get

$$T = 1 - \frac{\log(1 - 0.001)}{\log(0.6)} = 1 - 0.001956 = 0.9980 \qquad ■■$$

This equation is pessimistic for VLSI. In later paragraphs we will look at other equations that, based on clustering of faults, give more favorable results. Nevertheless, this equation illustrates an important concept. Test cost is not a linear function. Experience indicates that test cost follows the curve illustrated in Figure 1.4.

This curve tells us that we reach a point where substantial expenditures provide only marginal improvement in testability. At some point, additional gains become exorbitantly expensive and may negate any hope for profitability of the product. However, looking again at Eq. (1.14), we see that the defect level is a function of both testability and yield. Therefore, we may be able to achieve a desired defect level by improving yield.

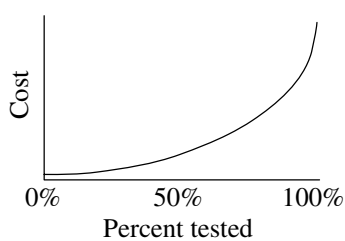


Figure 1.4 Typical cost curve for testing.

Example Yield is improved to $Y = 70\%$; what percentage of testing must be achieved to hold DL below 0.1%?

$$T = 1 - \frac{\log(1 - 0.001)}{\log(0.7)} = 1 - 0.0028 = 0.9972 \qquad \blacksquare\blacksquare$$

Equations (1.13) and (1.14) give the same results at the endpoints, but slightly different results between the endpoints. To understand why, it is necessary to look at the assumptions behind the derivations. Wadsack assumes that $yi = (1 - y)^i$, where yi represents the chips with i faults and y represents the actual functional yield. Williams and Brown assume the existence of n faults, that all faults have equal probability P_n of occurrence, and that the number of chips with i faults is

$$\binom{n}{i}(1 - P_n)^{n-i} P_n^i$$

Working out the derivations from these different starting points results in the different equations. However, regardless of which equation is used, the key point is that, in order to achieve an acceptable quality level AQL ($= 1 - DL$), the fault coverage has to be nearly perfect. In the words of Williams and Brown, the equations are intended to "give estimates for quick calculations." Wadsack, in his paper, points out that even in a circuit with 100% fault coverage, a failure occurred on the tester after the point where the test program had achieved 100% coverage of the faults. But then he points out that, in general, his derivation tends to be pessimistic.

Other authors have found the equations to be pessimistic; that is, even with fault coverage significantly less than that required by the equations, the quality level is better than predicted by the equations. For instance, Wiscombe[17] states that the Williams–Brown model "predicts higher defect levels than seen in practice." Maxwell et al. point out that for a defect level of less than 0.1%, the Williams–Brown equation required fault coverage in excess of 99.6%. However, they were able to realize those defect levels with about 96% fault coverage.[18]

The question of fault coverage versus defect levels was studied by Agrawal et al. in 1982.[19] Their study was motivated by the observation that the defect level equations "produced satisfactory results for chips with high yield (typically, SSI and MSI), but the predictions were too pessimistic for larger chips with lower yield." The authors hypothesize the existence of n faults for a faulty chip, and then examine the consequences of that assumption. They derive the following equation:

$$r(f) = \frac{(1 - f)(1 - y)e^{-(n_0 - 1)f}}{y + (1 - f)(1 - y)e^{-(n_0 - 1)f}} \tag{1.16}$$

In this equation, y is the yield, n_0 is the average number of faults on a faulty chip, f is the fault coverage, and $r(f)$ is the field reject rate for f. If the fault coverage is held fixed, then the defect level goes down as n_0 increases. The papers cited here suggest that the value $n_0 = 3$ appears to give reasonably good results at predicting defect level.

The model that was used to develop Eq. (1.16), referred to as the JSCC model, was subsequently refined using what the authors called the CAD model.[20] A Poisson

distribution is assumed for the faults, and the number of defects is assumed to have a clustered negative binomial distribution. With those assumptions the authors derived a reject ratio $r(f) = [y(f) - y]/y$, where

$$y(f) = [(1 + Ab(1 - e^{-cf})]^{-a} \tag{1.17}$$

In this equation, A is the chip area, f is the fault coverage, and a, b, and c are model parameters that are estimated by fitting $y(f)$ versus f to the experimental data.

In yet another derivation,[21] presented at a workshop in Springfield, Massachusetts, and referred to as the SPR model, the reject ratio $r_n = (y_n - y)/y_n$ is computed as a function of the yield y_n, after n vectors, and the true yield y. The variables y_n and y are computed as a function of the number of chips tested, the number of applied vectors, and the number of chips failing at vector i. The authors point out that the required data are derived from wafer probe. The calculations do not depend on estimated fault coverage of the test vectors. In this same study[21] the authors compare the five models for defect level estimation.

Comparison of the five models was done by gathering statistics on a high-volume chip at Delco Electronics. The chip was a 3-micron digital CMOS IC with 99.7% fault coverage. The test program consisted of 12,188 clock periods, and the cumulative fault coverage was computed after each vector. Of the 72,912 die initially considered, 847 chips that failed parametric test and 7699 chips that failed continuity test were removed from consideration. Of the remaining 64,366 chips, 18,476 failed the functional test. This resulted in an apparent yield of 71.30%. The true yield, using the SPR model, was estimated to be 70.92%. The results of the comparison are presented in Table 1.2.

In most columns the spread between these formulas varies by as much as a factor of two. The one exception is the last column, where the SPR and JSSC models differ by an order of magnitude. The bottom row of the table lists the actual fraction of defects detected at various stages of testing the chips. For the rightmost column, corresponding to a fault coverage of 99.70%, all the vectors had been applied, so no additional defects were found. However, each of the models predicts that additional tester escapes will occur.

TABLE 1.2 Comparing Yield

Model	Fault Coverage						
	20%	50%	80%	91%	95%	98%	99.70%
SPR	0.11291	0.08005	0.03531	0.02160	0.00927	0.00702	0.00532
JSSC	0.21383	0.11373	0.03730	0.01548	0.00834	0.00362	0.00048
CAD	0.21714	0.12439	0.04556	0.01985	0.01090	0.00432	0.00064
Wadsack	0.23267	0.14542	0.05817	0.02617	0.1454	0.00582	0.00087
Williams	0.24038	0.15788	0.06642	0.03046	0.01704	0.00685	0.00103
Actual	0.18440	0.08340	0.02830	0.01330	0.00740	0.00210	0

Although the Williams–Brown model tends to be the least accurate, at least for the data in this experiment, it appears to be the most popular, based on frequency of appearance in the literature. This may be due in large part to its simplicity, which makes it easy for engineers to explain the relationship between quality, process yield, and fault coverage. Perhaps, more significantly, any of these models can tell the user when the fault coverage must be improved. For example, if the user wants no more than 1000 defects per million (DPM), then all of these models convey the message that 98% fault coverage is insufficient.

The SPR model computes tester escapes without benefit of fault simulation. A drawback to this approach is the fact that, without fault coverage estimates for the test program, it could require several iterations on the test floor acquiring data before the test program is adequate. By contrast, when developing a test program with the aid of fault coverage estimates, it is more likely that the test will be at, or near, required coverage levels before it is used on the test floor.

Up to this point, when talking about fault coverage, the number used in the calculations was simply the number of modeled faults that were detected, divided by the total number of modeled faults. It has been assumed, for a given test coverage, that the coverage is uniform across the circuit. However, that may not be the case. Consider the test for a large chip, consisting of several functions. The test program may be a concatenation of several smaller test programs, each of which targets a single function. Suppose there are six clearly identifiable functions on the chip, then there might be six distinct test programs targeting the individual functions. The tests for five of the functions may be near 100%, while the test for the remaining function may be closer to 70%. Gross defects that might be detected in the other functions could escape detection in the function with low coverage.

Maxwell[22] showed that it is necessary to get a uniformly high coverage across the entire area of the chip. Also worth noting is the fact that each function may have some unique characteristics. For example, one function may be sensitive to noise. Another may use unique elements from a standard library, one or more of which are prone to failure. Conceivably a latch or flip-flop, for whatever reason, may have difficulty holding a particular state. These properties may not all be adequately addressed in one or more of the test programs.

Other investigations of defect levels have been performed. McCluskey and Buelow introduce the term *test transparency* (*TT*).[4] It is the fraction of all defects that are not detected by a test procedure:

$$TT = \text{defects not detected} / \text{total no. defects} = 1 - m/n$$

where n is the total number of defects and m is the number of defects detected. They show that, for $DL \leq 0.1\%$ and $Y \geq 90\%$, $DL = TT \cdot (1 - y)$. They state that it is customary to estimate test transparency by the percentage of single-stuck faults that are not detected by the test, $TT \geq 1 - T$, where T is the test coverage. Using $1 - T$ as an estimate for TT gives $DL = (1 - T) \cdot (1 - y)$, which is the Wadsack equation developed in 1978.

1.8 THE ECONOMICS OF TEST

In previous sections we examined some factors that affect the quality of test programs. In this section we examine factors that influence the cost of test. Quality and test costs are related, but they are not inverses of one another. As we shall see, an investment in a higher-quality test often pays dividends during the test cycle.

Test related costs for ICs and PCBs include both time and resource. As pointed out in previous sections, for some products the failure to reach a market window early in the life cycle of the product can cause significant loss of revenue and may in fact be fatal to the future of the product. The dependency table in Figure 1.5 shows test cost broken down into four categories[23]—some of which are one-time, non recurring costs whereas others are recurring costs. Test preparation includes costs related to development of the test program(s) as well as some potential costs incurred during design of the design-for-test (DFT) features. DFT-related costs are directed toward improving access to the basic functionality of the design in order to simplify the creation of test programs.

Many of the factors depicted in Figure 1.5 imply both recurring and nonrecurring costs. Test execution requires personnel and equipment. The tester is amortized over individual units, representing a recurring cost for each unit tested, while costs such as probe cards may represent a one-time, nonrecurring cost. The test-related silicon is a recurring cost, while the design effort required to incorporate testability enhancements, listed under test preparation as DFT design, is a nonrecurring cost.

The category listed as imperfect test quality includes a subcategory labeled as *tester escapes*, which are bad chips that tested good. It would be desirable for tester escapes to fall in the category of nonrecurring costs but, regrettably, tester escapes

		Personnel cost	Test card cost	Probe cost	Probe life	Depreciation	Volume	Tester setup time	Tester capital cost	Wafer radius	Die area	Wafer cost	Defect density
Test preparation	Test generation	*									*		
	Tester program	*											
	DFT design	*									*		
Test execution	Hardware		*	*	*	*	*						
	Tester	*				*	*	*	*				*
Test related silicon										*	*	*	
Imperfect test quality	Escape						*				*		*
	Lost performance						*						
	Lost yield						*						*

Figure 1.5 Cost/benefit dependencies of DFT.

are a fact of life and occur with unwelcome regularity. Lost performance refers to losses caused by increases in die size necessary to accommodate DFT features. The increase in die size may result in fewer die on a wafer; hence a greater number of wafers must be processed to achieve a given throughput. Lost yield is the cost of discarding good die that were judged to be bad by the tester.

The column in Figure 1.5 labeled "Volume" is a critical factor. For a consumer product with large production volumes, more time can be justified in developing a comprehensive test plan because development costs will be amortized over many units. Not only can a more thorough test be justified, but also a more efficient test—that is, one that reduces the amount of time spent in testing each individual unit. In low-volume products, testing becomes a disproportionately large part of total product cost and it may be impossible to justify the cost of refining a test to make it more efficient. However, in critical applications it will still be necessary to prepare test programs that are thorough in their ability to detect defects.

A question frequently raised is, "How much testing is enough?" That may seem to be a rather frivolous question since we would like to test our product so thoroughly that a customer never receives a defective product. When a product is under warranty or is covered by a service contract, it represents an expense to the manufacturer when it fails because it must be repaired or replaced. In addition, there is an immeasurable cost in the loss of customer goodwill, an intangible but very real cost, not reflected in Figure 1.5, that results from shipping defective products.

Unfortunately we are faced with the inescapable fact that testing adds cost to a product. What is sometimes overlooked, however, is the fact that test cost is recovered by virtue of enhanced throughput.[24] Consider the graph in Figure 1.6. The solid line reflects quality level, in terms of defects per million (DPM) for a given process, assuming no test is performed. It is an inverse relationship; the higher the required quality, the fewer the number of die obtainable from the process. This follows from the simple fact that, for a given process, if higher quality (fewer DPM) is required, then feature sizes must be increased. The problem with this manufacturing model is that, if required quality level is too high, feature sizes may be so large that it is impossible to produce die competitively. If the process is made more aggressive, an increasing number of die will be defective, and quality levels will fall. Point A on the graph corresponds to the point where no testing is performed. Any attempt to shrink the process to get more units per wafer will cause quality to fall below the required quality level.

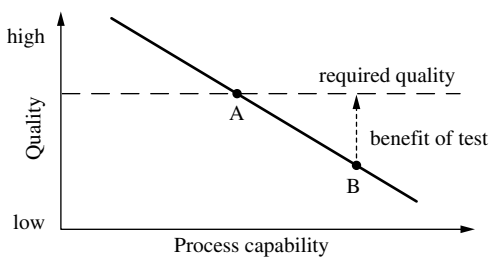


Figure 1.6 The benefits of test.

However, if devices are tested, feature sizes can be reduced and more die will fit on each wafer. Even after the die are tested and defective die are discarded, the number of good die per wafer exceeds the number available at the larger feature sizes. The benefit in terms of increasing numbers of good die obtainable from each wafer far outweighs the cost of testing the die in order to identify those that are defective.

Point B on the graph corresponds to a point where process yield is lower than the required quality level. However, testing will identify enough defective units to bring quality back to the required quality level. The horizontal distance from point A to point B on the graph is an indication of the extent to which the process capability can be made more aggressive, while meeting quality goals. The object is to move as far to the right as possible, while remaining competitive. At some point the cost of test will be so great, and the yield of good die so low, that it is not economically feasible to operate to the right of that point on the solid line.

We see therefore that we are caught in a dilemma: Testing adds cost to a product, but failure to test also adds cost. Trade-offs must be carefully examined in order to determine the right amount of testing. The right amount is that amount which minimizes total cost of testing plus cost of servicing or replacing defective components. In other words, we want to reach the point where the cost of additional testing exceeds the benefits derived. Exceptions exist, of course, where public safety or national security interests are involved.

Another useful side effect of testing that should be kept in mind is the information derived from the testing process. This information, if diligently recorded and analyzed, can be used to learn more about failure mechanisms. The kinds of defects and the frequency of occurrence of various defects can be recorded and this information can be used to improve the manufacturing process, focusing attention on those areas where frequency of occurrence of defects is greatest.

This test versus cost dilemma is further complicated by "time to market." Quality is sometimes seen as one leg of a triangle, of which the other two are "time to market" and "product cost." These are sometimes posited as competing goals, with the suggestion that any two of them are attainable.[25] The implication is that quality, while highly desirable, must be kept in perspective. *Business Week* magazine, in a feature article that examined the issue of quality at length, expressed the concern that quality could become an end in itself.[26]

The importance of achieving a low defect level in digital components can be appreciated from just a cursory look at a typical PCB. Suppose, for example, that a PCB is populated with 10 components, and each component has a defect level $DL = 0.999$. The likelihood of getting a defect free board is $(0.999)^{10} = 0.99004$; that is, one of every 100 PCBs will be defective—and that assumes no defects were introduced during the manufacturing process. If several PCBs of comparable quality go into a more complex system, the probability that the system will function correctly goes down even further.

Detecting a defective unit is often only part of the job. Another important aspect of test economics that must be considered is the cost of locating and replacing defective parts. Consider again the board with 10 integrated circuits. If it is found to be defective, then it is necessary to locate the part that has failed, a time-consuming and

error-prone operation. Replacing suspect components that have been soldered onto a PCB can introduce new defects. Each replaced component must be followed by retest to ensure that the component replaced was the actual failing component and that no new defects were introduced during this phase of the operation. This ties up both technician and expensive test equipment. Consequently, a goal of test development must be to create tests capable of not only detecting a faulty operation but to pinpoint, whenever possible, the faulty component. In actual practice, there is often a list of suspected components and the objective must be to shorten, as much as possible, that list.

One solution to the problem of locating faults during the manufacturing process is to detect faulty devices as early as possible. This strategy is an acknowledgment of the so-called *rule-of-ten*. This rule, or guideline, asserts that the cost of locating a defect increases by an order of magnitude at every level of integration. For example, if it cost N dollars to detect a faulty chip at incoming inspection, it may cost $10N$ dollars to detect a defective component after it has been soldered onto a PCB. If the component is not detected at board test, it may cost 100 times as much if the board with the faulty component is placed into a complete system. If the defective system is shipped to a customer and requires that a field engineer make a trip to a customer site, the cost increases by another power of 10. The obvious implication is that there is tremendous economic incentive to find defects as early as possible.

This preoccupation with finding defects early in the manufacturing process also holds for ICs.[27] A wafer will normally contain test circuits in the scribe lanes between adjacent die. Parametric tests are performed on these test circuits. If these tests fail, the wafer is discarded, since these circuits are far less dense than the circuits on the die themselves. The next step is to perform a probe test on individual die before they are cut from the wafer. This is a gross test, but it detects many of the defective die. Those that fail are discarded. After the die are cut from the wafer and packaged, they are tested again with a more thorough functional test. The objective? Avoid further processing, and subsequent packaging, of die that are clearly defective.

1.9 CASE STUDIES

Finally, we present the results of two studies into test thoroughness versus AQL and the consequences of decisions made with respect to test. The first is a classic study published in 1985 that serves to underscore the importance of achieving high fault coverage. The second is a study into the economics of multi-chip modules (MCMs). A model was created and parameters were varied in order to discern their effect on total product cost.

1.9.1 The Effectiveness of Fault Simulation

In this study, the results of which are shown in Figure 1.7, the authors were concerned with the fact that at 96.6% fault coverage they were still getting too many field rejects, and the costs of packaging and test were excessive.[4,28] A decision was made to improve the test program and determine what impact that would have on the defect level.

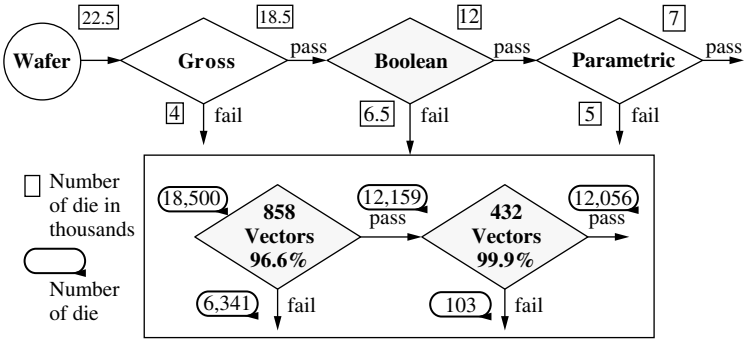


Figure 1.7 Fallout during test.

In their study, investigators analyzed 22,506 die. Of these, 4006 were eliminated at the start of testing because of failures due to gross defects, including opens, shorts, and so on. Then, 18,500 die were subjected to a functional test. The initial test consisted of 858 vectors that provided 96.6% fault coverage. This test identified 6341 failing devices. Over time, the initial test was increased to 992 vectors to address specific field reject problems encountered during production. During this study the test was enhanced by the addition of another 298 vectors to bring the total vector count to 1290. During their experiment, investigators recorded the vector number at which failures occurred. The original 858 vectors uncovered 6341 defective chips. The added 432 vectors uncovered an additional 103 defective chips.

1.9.2 Evaluating Test Decisions

The second study examined test decisions involving (MCMs). The MCM is a hybrid manufacturing technique in which several ICs are placed on an intermediate level of packaging. It can be used to package incompatible technologies such as CMOS and TTL, or it can be used to package digital circuits together with analog circuits that can't tolerate the noise generated by digital circuits. It can also be used to package digital circuits together with memory, such as cache memory, or it can be used to package two digital circuits that are either (a) too big to be placed on a single chip with existing technology or (b) those in which yield of a single, larger chip may be unacceptable. In this last instance, the MCM may be an intermediate phase until manufacturing advances permit the individual digital chips to be integrated onto a single die.

MCMs are often manufactured using known good die (KGD). The KGD is a bare die that has gone through extensive testing. In a normal flow, wafer sort is performed on individual die before they have been cut from the wafer. This is a test whose purpose is to identify, as quickly as possible, those die that are grossly defective. Then, those die that pass the test at wafer sort are packaged and tested more thoroughly. By contrast, KGD must be thoroughly tested on the wafer because they will be sold as

bare die, and the buyer will mount them directly onto the MCM without benefit of an additional layer of packaging. As a consequence of this approach, the MCMs that use these die must be processed in a clean room, which adds to manufacturing cost.

The cost of manufacturing MCMs is affected in significant ways by choices made with regard to test. Some of the factors include: chip yield and the thoroughness of test, the number of chips on the MCM, yield of the interconnect structure, yield of the bonding and assembly processes, and effectiveness of test and rework for detecting, isolating, and repairing defective modules. The High-Level Test Economics Advisor (Hi-TEA) evaluates decisions made with respect to these and other factors, including cost of materials and processes, yield parameters, and test parameters.[29] The metrics used by Hi-TEA are cost and quality: Hi-TEA attempts to optimize one while the other serves as a constraint.

The Hi-TEA user enters many parameters and/or assumptions into the system. Some of these inputs are easily obtained, such as the cost of labor and materials used to package and test the MCMs. Other costs are initially guesses, which can be refined as experience accumulates. In the paper cited here, the authors included several tables contrasting MCM cost versus chip AQL. One of the interesting results brought out was the trade-offs required to compensate for poor quality level of ICs used to populate the MCMs in some of their examples. It was also interesting to note that as AQL for the chips increased from 80% to 99.9%, total cost for MCMs followed a bell-shaped curve, first increasing, then decreasing, so that with 99.9% AQL, it cost less to manufacture MCMs that met a given AQL goal. Another byproduct of higher chip AQL was a significant reduction in the number of defective MCMs shipped to customers.

Figure 1.8 provides a summary of test cost versus quality trade-offs for several different test and DFT strategies. The test vehicle for this study was an MCM that contained a CPU, a coprocessor, and ten 4-Mbit SRAM chips. The clock speed for this MCM was faster than that of any existing workstations at the time of the design. It was assumed that there would be three defects per square inch for the CMOS CPU and coprocessors, and six defects per square inch for the BICMOS SRAM wafers. It

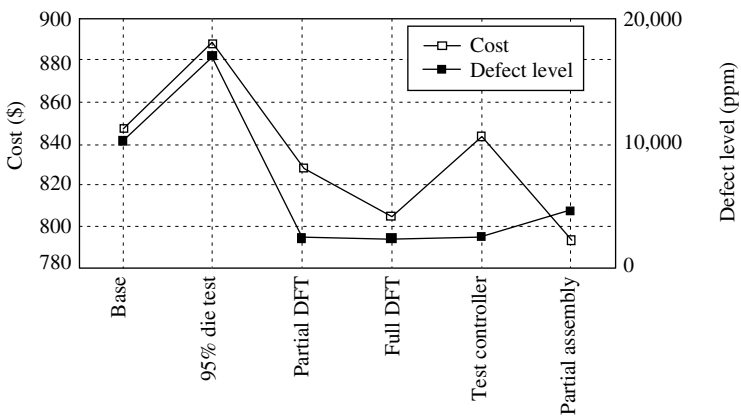


Figure 1.8 Cost/quality trade-offs for various test/DFT strategies.

was also assumed that 10% of the die would fail during burn-in. Test coverage at wafer probe was 80%, and coverage at the die level was 99%. Substrate yield was 99.999% and test coverage for MCM test was 95% of all possible defects, including faulty die, assembly errors, and so on.

From the base test, the next case reduced by half the test time for the die. As a result, the fault coverage for the die decreased from 99% to 95%. From Figure 1.8 it can be seen that, compared to the base case, final product cost increased by about 5% and defect level went up by almost 70%.

The next objective was to study the impact of DFT and built-in self-test (BIST) on the cost and quality of the MCMs. The first experiment involved adding DFT and BIST to the CPU and coprocessor. Compared to the base case, the use of partial DFT reduced defect level from 10,000 to about 3000 ppm while reducing cost from $845 to about $830. For the full DFT case the defect level remained about the same as with the partial DFT case, but cost fell to about $805. An advantage that did not get factored into these computations is the availability of the DFT features at higher levels of integration, such as systems test.

The use of a test controller on the MCM was intended to evaluate the situation where the manufacturer has no control over the ICs used in the design. In this scenario, the test controller provides greater access to the individual chips on the MCM. The cost of the additional test controller chip added $60 to the cost of the MCM, but its presence helped to reduce the overall test cost slightly when compared to the base case. The defect level was reduced by almost 80% relative to the base case.

The final scenario considered testing the MCM after the SRAMs were attached. If defects were encountered, they were repaired and the MCM retested. Then, when the partial assembly passed the test, the CPU and coprocessor were mounted and the MCM was retested. In this scenario the SRAMs can be considered hardcore (cf. Section 9.7.1) and used to test the remaining logic on the MCM. Because diagnosis is improved, it is less expensive to isolate defects and make repairs. Special fixtures can be created to improve access to test points on the MCM. Note that this case provides the lowest overall cost of the MCM, although the defect level is slightly higher than when DFT is used.

1.10 SUMMARY

During the past three decades a great deal of research has gone into the various facets of IC design, including system architectures, equipment used to create digital circuits with ever-shrinking feature sizes, and EDA tools used to facilitate the migration from concept to digital product. Along the way, quality has benefited from a better understanding of defect mechanisms, the development of better test methods to identify and diagnose the causes of defects, and a better understanding of the technical and economic trade-offs required to achieve desired quality levels.

Product reliability is another beneficiary as digital products have migrated from SSI (small-scale integration), through very-large-scale integration (VLSI), into deep

submicron (DSM). Greater integration has resulted in fewer assembly steps and fewer soldering joints. As far back as 1979 it was reported that, based on five billion device hours of experience, LSI devices with 70 to 100 gates per chip experienced twice the failure rate of SSI devices with four to eight gates per chip. Put another way, LSI devices experienced one-seventh the failure rate of SSI devices, on a per-gate basis.[30] CMOS technology, running at much lower power levels than equivalent circuits implemented in previous technologies (ECL, TTL, etc.), has contributed to improved reliability.

As the IC industry matures, and engineers gain a better understanding of the many factors that contribute to yield loss, they are able to apply this new-found knowledge to reduce both the sizes and the numbers of defects that occur in a given die area, with the result that yields increase. This is all the more remarkable in view of the fact that feature sizes continue to shrink and chip complexity continues to increase. A relationship between complexity and minimum defect size is suggested in Figure 1.9, where trends are projected to the year 2010.[31]

The incentive to shrink die size is motivated by a rather basic imperative, improved profitability.[32] Consider a wafer with N die and a yield Y. There will be $Y \times N$ good die on the wafer. Each of these will be sold for Z dollars, producing an income of $Y \times N \times Z$. This income must exceed the cost of designing, manufacturing, packaging, testing, and marketing the chips. If die size is reduced, there will be more die on each wafer, but the number of bad die may increase. If shrinking the die size causes a disproportionately larger increase in the number of good die, then income increases, assuming production costs do not go up disproportionately. Given a fixed selling price, then, the object is to find die size and yield that maximize the product term $Y \times N \times Z$.

A simplistic analysis could lead to the conclusion that the number of good die must increase disproportionately. Consider the following: If there were simply a fixed number of point defects on a wafer, and they caused $(1 - Y)$ die to fail, then doubling the number of die on a wafer would produce $N + (1 - Y) \times N$ good die. In effect, the overall yield increases. However, it is not quite that simple.

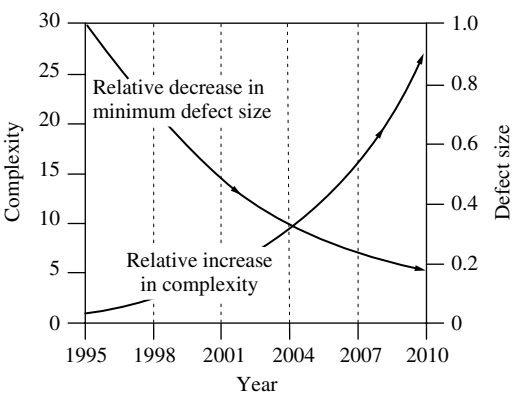


Figure 1.9 Complexity versus defect size.

As feature sizes shrink, supply voltages are reduced. This reduces power consumption, heat dissipation, and failures caused by electric fields greater than the circuit can tolerate. But, reducing the supply voltage increases gate delay and thus reduces the maximum clock rate. To compensate for this, the threshold voltage (the voltage at which the transistor turns on) is reduced. If the threshold voltage is reduced too far, leakage current becomes excessive. It is estimated that for every 60 mV that the threshold is lowered, leakage current increases by an order of magnitude.[33] New failure mechanisms may be introduced into the process. Lower operating voltages imply less noise margin. Traces on the die are closer together, resulting in greater potential for crosstalk. Greater capacitive coupling exists. Also, some point defects on the wafer that may not have been problems at larger feature sizes may become problems as feature sizes are reduced.

In summary, processes are improving, but as long as the universe is subject to entropy, defects will continue to occur. The existence of defects implies a need for test programs capable of detecting them, whether it be for reducing field rejects or to help debug first silicon. The existence of chips with larger gate count implies a need to develop more efficient test programs. The emergence of new fault mechanisms implies a need for new test algorithms targeting those fault mechanisms. Furthermore, the ability to accurately compute defect level is important because it tells us that, given levels of testability and yield beyond which we cannot hope to improve (economically), we must expect a certain percentage of defective units shipped and plan our business strategy accordingly, whether it be to stock more spare parts or to improve our service department.

Another factor that has grown in importance in recent years is end-user expectations. In 1994, when a floating point problem was encountered in early Pentium processors, the first inclination by Intel Corp. was to downplay the significance of the problem, asserting that a typical user might only encounter an incorrect calculation once every 27 years. The outcry far exceeded anything that was anticipated by Intel. They found that in order to maintain a favorable public image, it was necessary to establish a generous return policy for anyone with a Pentium based microprocessor system. The resulting message from this experience is that, with electronic products more pervasive than ever in many different end-user products, there is a less forgiving public unwilling to understand or tolerate defective products. One slip by a major vendor, and there will be another company waiting in the wings, ready to step in and exploit the opportunity.

It is interesting to note that the delivery of correct and reliable computing is influenced by factors that can be classified as nontechnical. For example, IBM's Server Group claims that the mean time between critical failures (MTBCF) of its System/390 mainframe is 20 to 30 years, where MTBCF is the average time between failures that force a reboot and initial program load.[34] A large part of the reason for this is because the core software is extremely stable, a change is implemented only if it is determined beyond all doubt that a bug exists. Of course, the hardware must also be stable.

One of the design parameters for a new system being developed is mean time before failure (MTBF). The goal is to keep a system up and running as long as possible. However, another parameter that often must be considered when developing a new

system is mean time to repair (MTTR). While it is desired not to have a system fail, in some circumstances it may be even more desirable to be able to get a system up and running again after it has failed. This may necessitate the inclusion of hardware whose sole purpose is to help diagnose and isolate failure to a field replaceable unit (FRU). Design-for-test or built-in self-test may be vitally necessary to achieve MTTR goals.

Change, and an urge for novelty, are key aspects of human existence, but sometimes these urges must be resisted. This ability to resist the urge to make changes unless it is absolutely necessary to do so is cited as a major reason for Intel's success. In an article in the *San Jose Mercury News*, the story is told of a drop in yield at one of Intel's foundries.[35] An investigation revealed that a processing change caused wafers to move more quickly from one station to the next. As a result, the temperature of the wafers as they arrived at the next station deviated from what it had previously been, and the deviation was enough to adversely affect the yield of the die on those wafers.

This drop in yield was notable because Intel reportedly practices a policy called "Copy Exactly." This practice involves building a fabrication plant as part of the research and development process for a new product. The R&D process involves not just the designers of a next generation chip, but also the people in manufacturing who must fabricate and test it. Once a manufacturing process is put into place, changes are not made until after considerable debate and considerable examination of the data. This is basically an implementation of *concurrent engineering*, which is defined as "a systematic approach to the integrated, concurrent design of products and their related processes, including manufacture and support."[36]

An appreciation for the relationship between test cost, yield, and reject rate can be gained by considering an analogous situation in the field of communications. When communicating through a noisy medium, communications can be made more reliable by increasing transmission power. However, Shannon's theorem for communications in a noisy channel tells us that it is possible to make the transmission error rate arbitrarily small by resorting to error correcting codes (ECC). The most economic solution is found by factoring in both the cost of transmission power and the cost of employing ECC circuitry to find a solution that allows the most reliable communication at the highest possible rate, at the lowest possible cost.

Consider that the objective, when processing wafers, is to ship only good die. If field reject rate is too high, it could be improved by resorting to larger feature sizes. However, it can also be improved by employing a more thorough test that identifies more of the defective die before they are shipped to customers. The most economic solution is a complex function of process yield and test coverage.

PROBLEMS

1.1 For a semiconductor process with a yield $Y = 0.7$, compute the defect level DL by means of Eqs. (1.13) and (1.14) for values of T equal to 0.7, 0.8, 0.9, and 0.975. Repeat using Eq. (1.16), with values of n_0 equal to 1 and 3. Repeat all calculations for $Y = 0.9$.

1.2 Assume that the relative cost, C_d, of diagnosing and repairing defects, expressed as a function of the percentage t of faults tested, is $C_d = 100 - 0.7t$. Furthermore, assume that the cost C_p of achieving a particular test percentage t is $C_p = \frac{t}{100 - t}$. What value of t will minimize total cost?

1.3 Using Eq. (1.14), draw a graph of defect level versus fault coverage using each of the following values of yield as a parameter: $Y = \{.40, .50, .70, .90, .95\}$.

1.4 Using Eq. (1.5), calculate $P(0)$ for $\lambda_0 = \{.25, .5, .75, 1.0, 2.0\}$. Repeat using Eq. (1.10) and assume $D_0A = \{.25, .5, .75, 1.0, 2.0\}$. Repeat using Eq. (1.11), for $\alpha = 2$ and for $\alpha = 4$.

1.5 Assume two randomly distributed defects per square inch, and assume that each defect only affects one die. If there are four die on each square inch of wafer, what is the yield? If feature sizes are shrunk so that there are nine die per square inch, what is the yield?

1.6 Assume that the maximum allowable reject rate for a particular IC is 500 ppm. Use Eq. (1.5) to draw a graph of yield versus fault coverage for values of $n_0 = 0, 1, 2, 3, 4, 5$.

1.7 Given an MCM with 20 die, each of which has an AQL of 99.5%, what is the probability of a fault-free MCM?

REFERENCES

1. Doyle, E. A. Jr., How Parts Fail, *IEEE Spectrum*, October 1981, pp. 36–43.
2. Williams, T. W., and N. C. Brown, Defect Level as a Function of Fault Coverage, *IEEE Trans. Comput.*, Vol. C-30, No. 12, December 1981, pp. 987–988.
3. Rechtin, Eberhardt, The Synthesis of Complex Systems, *IEEE Spectrum*, July 1997, Vol. 34, No. 7, pp. 51–55.
4. McCluskey, E. J. and F. Buelow, IC Quality and Test Transparency, *Proc. Int. Test Conf.*, 1988, pp. 295–301.
5. Donlin, Noel E., Is Burn-in Burned Out?, *Proc. Int. Test Conf.*, 1991, p. 1114.
6. Henry, T. R., and Thomas Soo, Burn-in Elimination of a High Volume Microprocessor Using I_{DDQ}, *Proc. IEEE Int. Test Conf.*, 1996, pp. 242–249.
7. Weber, Samuel, Exploring the Time to Market Myths, *ASIC Technol. News*, Vol. 3, No. 5, September 1991, p. 1.
8. Teichrow, D., and E. A. Hershey, III, PSL/PSA: A Computer-Aided Technique for Structured Documentation and Analysis of Information Processing Systems, *IEEE Trans. Software Eng.*, Vol. SE-3, No. 1, January 1977, pp. 41–48.
9. Bell, C. G., and A. Newell, *Computer Structures: Readings and Examples*, McGraw-Hill, New York, 1971.

10. Davis, A. M., and D. A. Leffingwell, Using Requirements Management to Speed Delivery of Higher Quality Applications, Technical Report 0001, Requisite, Inc., http://www.requirement.com/requisite.
11. Sanborn, J. L., Evolution of the Engineering Design System Data Base, *Proc. 19th D.A. Conf.*, 1982, pp. 214–218.
12. Cunningham, J. A., The Use and Evaluation of Yield Models in Integrated Circuit Manufacturing, *IEEE Trans. Semicond. Mfg.*, Vol. 3, No. 2, May 1990, pp. 60–71.
13. Seeds, R. B., Yield and Cost Analysis of Bipolar LSI, *Proc. IEEE IEDM*, Washington, D.C., October 1967.
14. Murphy, B. T., Cost-Size Optima of Monolithic Integrated Circuits, *Proc. IEEE*, Vol. 52, December 1964, pp. 1537–1545.
15. Stapper, C. H., Defect Density Distribution for LSI Yield Calculations, *IEEE Trans. Electron Devices*, Vol. ED-20, July 1973, pp. 655–657.
16. Wadsack, R. L., Fault Coverage in Digital Integrated Circuits, *Bell Syst. Tech. J.*, May–June 1978, pp. 1475–1488.
17. Wiscombe, Paul C., A Comparison of Stuck-at Fault Coverage and I_{DDQ} Testing on Defect Levels, *Proc. Int. Test Conf.*, 1993, pp. 293–299.
18. Maxwell, P. C., R. C. Aitken, V. Johansen, and I. Chiang, The Effect of Different Test Sets on Quality Level Prediction: When Is 80% Better than 90%?, *Proc. Int. Test Conf.*, 1991, pp. 358–364.
19. Agrawal, V. D., S. C. Seth, and P. Agrawal, Fault Coverage Requirement in Production Testing of LSI Circuits, *IEEE J. Solid-State Circuits*, Vol. SC-17, No. 1, February 1982, pp. 57–61.
20. Das, D. V., S. C. Seth, P. T. Wagner, J. C.Anderson, and V. D. Agrawal, An Experimental Study on Reject Ratio Prediction for VLSI Circuits: Kokomo Revisited, *Proc. 1990 Int. Test Conf.*, pp. 712–720.
21. Seth, S. C. and V. D. Agrawal, On the Probability of Fault Occurrence, in *Defect and Fault Tolerance in VLSI Systems*, ed. I. Koren, pp. 47–52, Plenum, New York, 1989.
22. Maxwell, Peter C., Reductions in Quality Caused by Uneven Fault Coverage of Different Areas of an Integrated Circuit, *IEEE Trans. CAD*, Vol. 14, No. 5, May 1995, pp. 603–607.
23. Wei, S., P. K. Nag, R. D. Blanton, A. Gattiker, and W. Maly, To DFT or Not to DFT?, *Proc. Int. Test Conf.*, 1997, pp. 557–566.
24. Aitken, R. C., R. K. Scudder, and P. C. Maxwell, Never Mind the Cost of Test—Look at the Value!, Test Cost Reduction Workshop, *SEMI 1997*, pp. D1–D5.
25. Young, Lewis H., *Electronic Business Today*, October 1995, p. 50.
26. *Business Week*, August 8, 1994.
27. Thompson, Tom, How to Make the World's Fastest CPUs, *Byte Magazine*, Vol. 22, No. 2, February 1997, pp. bona3–bona12.
28. Daniels, R. G., and W. C. Bruce, Built-In Self-Test Trends in Motorola Microprocessors, *IEEE Des. Test, Comput.,* April 1985, Vol. 2, No. 2, pp. 64–71.
29. Abadir, M. S., et al., Analyzing Multichip Module Testing Strategies, *IEEE Des. Test Comput.*, Spring 1994, Vol. 11, No. 1, pp. 40–52.
30. Slana, Matthew F., Workshop Report: Computer Elements for the 80's, *IEEE Comput.*, Vol. 12, No. 4, April 1979, p. 102.
31. Vallett, David P., IC Failure Analysis: The Importance of Test and Diagnostics, *IEEE Des. Test*, July–September 1997, Vol. 14, No. 3, pp. 76–82.

32. Oldham, William G., The Fabrication of Microelectronic Circuits, *Sci. Am.*, September 1977, Vol. 237, No. 3, pp. 111–128.
33. Pountain, Dick, Amending Moore's Law, *Byte Magazine*, March 1998, pp. 91–95.
34. Halfhill, Tom R., Crash-Proof Computing, *Byte Magazine*, April 1998, pp. 60–74.
35. Gillmor, Dan, Curb on Tweaking Made Intel Strong, *San Jose Mercury News*, August 18, 1997, p. 1E.
36. Carter, Donald E., and B. S. Baker, Concurrent Engineering: The Product Development Environment for the 1990s, *Addison-Wesley, Reading, MA*, 1992.

CHAPTER 2

Simulation

2.1 INTRODUCTION

Simulation is an imitative process. It is used to study relationships between parameters that interact in a system. In some cases it may point out errors that cause a design to respond incorrectly. In other cases it permits optimization of a design for maximum performance or economy of operation or construction. In still other situations, the system may be so complex that simulation is the only way that variables affecting the design, and their interaction with each other, can be controlled and studied.

In order to imitate the behavior of a product or system, simulation employs models. A model is an imperfect replica. It must contain enough information to accurately represent the behavior of the variables of interest in the process or system being studied, but must not be so complex as to obscure details of the variables and their relationships or so intricate that its cost approaches that of simply building the device or system to be studied.

This chapter will focus on methods used to simulate digital logic circuits in order to predict their behavior in the presence of various stimuli and environmental factors. Note that the accuracy of the prediction of circuit response depends on the accuracy and level of detail of the circuit model provided to the simulator. In future chapters we will examine fault simulation and other methods for verifying correctness of designs and correctness of the fabricated product. Much can be learned by comparing and contrasting methodologies used in simulation, and fault simulation, with those used in design verification. In fact, as circuits get larger and more complex, the arguments for integrating design and test activities become more compelling. To the extent that the design effort can be leveraged in the manufacturing test development task, the overall development cost for design and test can be reduced.

2.2 BACKGROUND

Early designers of digital logic implemented their circuits on printed circuit boards (PCBs) using integrated circuits (ICs) characterized as small-scale integration (SSI),

Digital Logic Testing and Simulation, *Second Edition*, by Alexander Miczo
ISBN 0-471-43995-9 Copyright © 2003 John Wiley & Sons, Inc.

medium-scale integration (MSI), and large-scale integration (LSI). Logic designers seldom simulated their designs. Rather, they created *prototype*s. After the prototype was debugged, layout of the PCB would begin. If design errors were discovered after the PCB was fabricated, the errors were repaired with wires that were color-coded to indicate an engineering change order (ECO).

The prototype is a physical mockup of the circuit being designed. Connections are made by wire wrap or other means that can be easily altered to correct design errors. It is used to evaluate logical correctness and, possibly, timing characteristics of a design. The prototype is attractive because it can run at or near design speed, it can be evaluated under actual operating conditions, it does not require detailed simulation models of the components used in the design, and it can be run with virtually unlimited amounts of stimuli. Various types of test equipment can be hooked up to the design to evaluate its performance, debug problems, and determine relative timing margins and voltage levels. If the system configuration includes operational software and diagnostic tests, development and debug of this software can begin on the prototype.

The prototype has its drawbacks. Many months of effort and great expenditure of resources may be required to build the prototype.[1] It normally accommodates only a single experiment at a time and a considerable amount of time may be required to set up experiments. If the prototype goes down for any length of time because of failure or damage to a critical part, the entire design team may be idled. Furthermore, with increasing amounts of logic being incorporated into single ICs, prototypes offer less insight into timing issues.

In the late 1970s, simulation began to play a more important role in IC design. Foundries emerged that accepted logic designs and converted them to working silicon. Much of the "glue" logic on PCBs that was implemented with SSI and MSI parts began to find its way into ICs. This led to PCBs that were less densely populated, requiring fewer manufacturing steps. As a result, PCBs became more economical to produce, and a welcome byproduct of this evolution was an increase in reliability.

The United States Department of Defense (DoD) recognized a problem in this migration to custom ICs. The DoD required that there be a second source for components used in digital circuits. Their concern was that a sole supplier might become financially insolvent, and critical components used in weapons systems would no longer be available. The advent of design tools and foundries capable of producing unique digital functions prompted the DoD to initiate the VHSIC (Very High Speed Integrated Circuit) program. The goal was to learn as much as possible about this coming revolution in digital design.

To address the problem of sole sources for digital circuits, the DoD determined that there would have to be a common language for describing digital designs. Then, when a supplier provided a digital circuit for a DoD system, if it were not a standard, off-the-shelf part that was available from two or more sources, the supplier would be required to provide a formal description in a language sanctioned by the DoD. To that end, DoD sponsored a conference at the Woods Hole Oceonographic Center in the summer of 1981. Many experts on hardware description languages (HDLs) met

to discuss the various aspects of HDLs. A number of these languages already existed. In fact, the IBM/360 family of computers had been described in APL (A Programming Language) in 1963.[2] Other HDLs appeared over the years, the most common of these being A Hardware Programming Language (AHPL),[3] which is based on APL, Computer Description Language (CDL),[4] and Digital Description Language (DDL).[5]

From VHSIC and the Woods Hole conference, VHSIC Hardware Description Language (VHDL) eventually emerged. At the same time that VHDL was being defined and refined, the Verilog HDL was emerging as a commercial product. Verilog was initially proprietary, but eventually became an open language. As a result, two widely accepted HDLs currently exist, and a large number of design and test tools based on these languages have appeared in the marketplace.

Simulators based on these two languages have benefited from numerous enhancements that have improved their efficiency, effectiveness, and ease of use. Simulators exist that can operate on models described at levels of abstraction ranging from switch level to behavioral. The behavioral descriptions can represent designs equivalent to hundreds of thousands up to millions of logic gates. Furthermore, these simulators can process circuits described at multiple levels of abstraction: part behavioral, part gate-level, and part switch-level. The simulators support creation of test stimuli with numerous constructs that provide flexible control of simulation, afford visibility into intermediate results generated during simulation, and include print and debug capabilities that enable the user to identify precisely where timing and/or behavior fail to meet specifications.

The prototype, though not as popular as it once was, nevertheless endures. Modern-day prototypes appear in the form of emulation systems made from field-programmable gate arrays (FPGAs).[6] These are used to evaluate large, complex designs that would take enormous amounts of time to simulate in software. With an emulator running at clock speeds of 5 to 10 MHz, performance gains of up to six orders of magnitude are possible over logic simulation on a workstation.

In a sense we have come full circle with the growing use of reusable macros, or *virtual components* (VC), which are analogous to the MSI and LSI components used in previous generation designs. The emphasis is on "reusable," meaning that the VC is a general function that can be stored in a library and pulled into almost any design. As an example, a counter may have parallel load, count-up and count-down capabilities. A user might then hard-wire the VC to perform only a count-up operation. An IC that is designed using VCs becomes a *system-on-a-chip* (SoC). The company that designs the SoC, sometimes called a *core module* or *drop-in function*, may not fabricate the design, but, rather, may make the design available to other companies in the form of RTL code. The other company then inserts or drops it into a larger design. Companies that sell these designs do not sell components, rather, they sell *intellectual property* (IP).

The behavior of these cores is usually described in Verilog and/or VHDL. A design team could conceivably create a fairly large design completely out of core modules, just as early designers connected SSI, MSI, and LSI components together. Since core modules are used by many customers, designers who use

them may feel comfortable in assuming that the cores are designed correctly and would focus their design effort on verifying the interconnects between two or more of these modules.

2.3 THE SIMULATION HIERARCHY

Digital systems can be described at levels of abstraction ranging from behavioral to geometrical. Simulation capability exists at all of these levels. The *behavioral description* is the highest level of abstraction. At this level a system is described in terms of the algorithms that it performs, rather than how it is constructed. The development of a large system may begin by characterizing its behavior at the behavioral level, particularly if it is a "first of a kind" (cf. Section 1.4). A goal of behavioral simulations is to reveal conceptual flaws.

When simulating behaviorally, the user is interested in determining things like optimum instruction set mix. This is done by studying the effects of sequences of instructions on data flow. Data flow through system elements can also be studied at this level in order to detect potential bottlenecks. For example, it serves no useful purpose to put a more powerful CPU into a system if the existing CPU is always waiting for data from a memory or I/O unit. Trade-offs between hardware and software can also be determined. If some software sequences are executed often, such as when servicing interrupt requests, performance might be improved by implementing the sequence in hardware. Partitioning, or modular decomposition, can also be performed at this level, to determine the best allocation of functions to modules. When behavioral simulations are complete, the behavioral model can serve as a specification for the system design.

Once the system has been specified, a *register transfer level (RTL)* model, sometimes referred to as a *functional* model, can be used to describe the flow of data and control signals within and between functional units. The circuit is described in terms of flip-flops, registers, multiplexers, counters, arithmetic logic units (ALUs), encoders, decoders, and elements of similar level of complexity. Data can be represented at various levels of abstraction, ranging from Booleans to complex numbers, or can be represented as ASCII strings. The building blocks and their controlling signals must be interconnected so as to function in a manner consistent with the preceding behavioral level description.

A *logic* model describes a system by means of switching elements or gates. At this level the designer is interested in correctness of designs intended to implement functional building blocks and units. Performance or timing of the design is a concern at this level. Closely related to the logic model is the *switch-level* model used to describe behavior of metal oxide semiconductor (MOS) circuits.[7] A switch-level network consists of nodes connected by transistors. Each node has value 0, 1, Z, or X and each transistor is open, closed, or indeterminate. Logic processing is augmented by capabilities needed to perform strength resolution when a node is driven by two or more MOS devices. The capacitance at a node may be sufficient to hold a charge after all drivers are turned off, so the node behaves like a latch. If this

property of MOS devices is recognized by a simulator, greater accuracy in predicting circuit behavior may be possible.

A *circuit* level model is used on individual gate and functional level devices to verify their behavior. It describes a circuit in terms of devices such as resistors, capacitors, and current sources. The simulation user is interested in knowing what kind of switching speeds, voltages, and noise margins to expect. Finally, the *geometrical* level model describes a circuit in terms of physical shapes.

Simulation at a high level of abstraction requires less detailed processing; hence simulation speed is greater and more input stimuli can be evaluated in a given amount of CPU time. In most cases the loss of detail is known and accepted. However, there are instances where the designer may be unaware that information is lost, information whose absence may obscure details essential to a proper understanding of the circuit's behavior. The importance of the information may depend on whether the product being designed is synchronous or asynchronous. In synchronous designs, clocking of bistable devices is usually controlled in such a way as to make them less susceptible to unexpected pulses caused by transient signals. In asynchronous designs, where designers have the freedom to create clock pulses for flip-flops and latches, circuits are more susceptible to erratic behavior.

2.4 THE LOGIC SYMBOLS

Test problems, as well as other circuit issues, are often described most effectively by means of schematic diagrams. Figure 2.1 introduces the logic symbols that are used in this text, together with truth tables describing their behavior. In these schematics the binary values, 0 and 1, are augmented with the values X and Z. X represents an unknown or indeterminate signal value, while Z represents a floating signal. A net assumes the value Z when it is not being driven by any logic element, it has effectively been disconnected from the circuit. In Figure 2.1(e), the tri-state element has the enabling input *En*. When $En = 1$ the tri-state element behaves like a buffer, and when $En = 0$ the tri-state output is disconnected from its input, regardless of what value appears at the input. That condition is represented by a Z on the output.

A small bubble or circle on an input, output, or enable of a logic element represents an inverted signal. For example, the inverters shown in Figure 2.1(b) complement the logic value applied at the input. On an enable signal, such as the tri-state buffer, a bubble indicates an active low enable, meaning that the output floats when the enable is high and input data passes through the tri-state device when the enable is low.

The inputs and outputs of logic functions are called *terminals* or *ports*. Any wire that connects two or more terminals is called a *net*. The term net will also apply to any set or collection of interconnected terminals. An input terminal that is physically accessible at an IC pin or logic board pin is called a *primary input*. An output terminal that is physically accessible is called a *primary output*. An output terminal of a logic function will also sometimes be called a *node*.

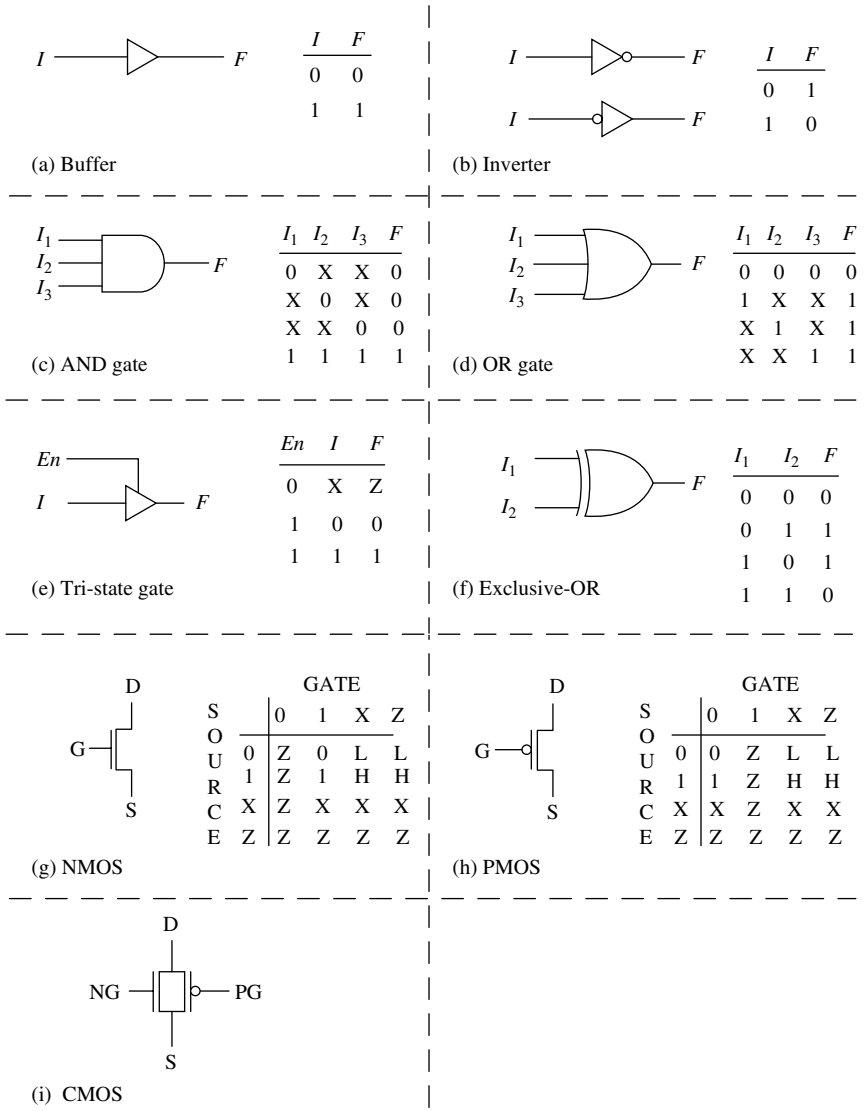


Figure 2.1 Some basic switching elements.

The AND circuit and the OR circuit are commonly referred to as *gates*. The AND, sometimes referred to as a *conjunction*, is high, or true, if all of its inputs are high. A low on any input to the AND circuit is called a *blocking signal*; it can block or gate out signals applied to other inputs, thus preventing them from passing through to the output. The OR, or *disjunction*, is low if all of its inputs are low. A logic 1 on any input to the OR is a blocking signal. Over time, the term gate has

come to embrace the other elements (Exclusive-OR, tri-state, etc.), even though their behavior as gates is not so evident.

An AND gate with a bubble on its output is a NAND gate. It has been known for almost a century that the NAND can be used to implement other logic functions.[8] The two-input NAND is often used as a measure of complexity for a circuit. For example, if the size of a function is described as being 20,000 gate equivalents, those 20,000 gates are understood to be two-input NAND gates.

Logic functions can be expressed in terms of MOS transistors. The basic building blocks are the NMOS and PMOS devices. The terminals are identified as S, G, and D, denoting source, gate, and drain. The transistor conducts when the gate is active. The NMOS device in Figure 2.1(g) conducts when the gate is at logic 1, and the PMOS device conducts when the gate is at logic 0. The symbol L denotes a value of 0 or Z at the drain, whereas H denotes a value of 1 or Z. The CMOS device has both negative gate (NG) and positive gate (PG). The values on these gates are normally the complement of one another. The CMOS device conducts when NG is 1 and PG is 0. The transistor level model is more accurate in terms of representing the actual physical structure of the circuit, but the level of detail may be so great as to obscure its basic functionality.

Logic operations can be described using Boolean equations. The equation

$$Z = A \cdot B + C \cdot \overline{D}$$

is called a *sum-of-products*, sometimes said to be in *disjunctive normal form*. A dot (·) indicates an AND operation, a plus (+) indicates an OR operation, and a bar above a variable indicates that it is complemented. The same logic operation can be described by

$$Z = (A + C) \cdot (B + C) \cdot (A + \overline{D}) \cdot (B + \overline{D})$$

This form is called a *product-of-sums*, also said to be in *conjunctive normal form*. For this logic operation the sum of products is more economical, requiring two AND gates and one OR gate, whereas the second expression requires four OR gates and one AND gate. For other logic functions the product of sums may be more economical.

2.5 SEQUENTIAL CIRCUIT BEHAVIOR

A generic sequential circuit is often represented by the Huffman model[9] in Figure 2.2. The circuit consists of a combinational part and feedback lines $Y_1, ..., Y_L$, which pass through delay elements $d_1, ..., d_L$ and then act as additional inputs to the combinational logic. The set of values $\{y_1, y_2, ..., y_L\}$ constitute the present state of the machine, while the values $\{Y_1, Y_2, ..., Y_L\}$ constitute the next state. Because there are a finite number of possible states, the circuit is called a *finite state machine*. The outputs z_i are a function

$$z_i = z_i(x_1, ..., x_n, y_1, ..., y_L)$$

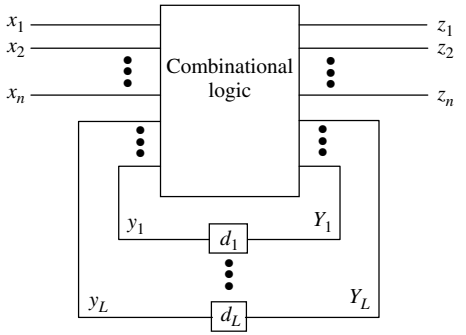


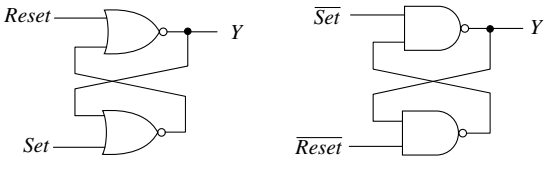
Figure 2.2 Huffman model.

of the values on the inputs and the present state. The delay elements d_1, ..., d_L may represent distributed delay inherent in the logic devices, they may represent lumped delay elements specifically designed to delay signals by some known fixed amount, they may be flip-flops controlled by one or more clock signals, or they may be composed of elements from each of these types. If the devices are all controlled by a common clock signal (or signals), then the circuit is *synchronous*; that is, its actions are synchronized by some external signal(s). If the delays are inherent in the devices, and not otherwise controllable by signals external to the circuit, the circuit is classified as *asynchronous*.

A circuit that has both clocked and unclocked delays may be placed in either category; the distinction often depends on the exact purpose of the asynchronous signals. A circuit in which memory devices can be asynchronously set or reset, but that is otherwise completely controlled by clock signals, is usually classified as synchronous. Sequential circuits are sometimes referred to as *cyclic*, a reference to the presence of feedback or closed loops, as distinguished from combinational circuits, which are termed *acyclic*. However, authors will also sometime distinguish between sequential cyclic and sequential acyclic circuits (cf. Section 5.4.1).

A frequently used memory element is the cross-coupled latch, implemented using either NOR gates or NAND gates, as depicted in Figure 2.3. These latches may appear by themselves or as constituent building blocks in other memory devices. The value on output Y at time t_{n+1} is determined by values on the *Set* and *Reset* input lines and by the present state of the latch. Given a present state y, and values on its *Set* and *Reset* inputs, the next state can be determined from a *state table* (cf. Figure 2.3). The value within the state table, at the intersection of a row corresponding to the present state and a column corresponding to the applied input value(s), specifies the next state to which the circuit will transition.

Entries containing dashes denote indeterminate states. For the NOR latch the column corresponding to (*Set,Reset*) = (1,1) contains dashes. It would be illogical to set and reset the latch simultaneously; and if the combination (1,1) were applied, followed by the combination (0,0), the final state of each such device appearing in the



Y \ SR	00	01	10	11
0	0	0	1	–
1	1	0	1	–

(a) NOR Latch

Y \ SR	00	01	10	11
0	–	1	0	0
1	–	1	0	1

(b) NAND Latch

Figure 2.3 Cross-coupled latches.

circuit would depend on the physical properties of that device. A similar consideration holds if the sequence {(0,0), (1,1)} were applied to the inputs of the NAND latch. A latch may be preceded by gates that permit it to be controlled by a clock. This is illustrated in Figures 2.4(a) and 2.4(b). In Figure 2.4(b) there is a single *Data* input whose value is inverted in one of two paths so the latch never sees the illegal input combination (0,0).

Clock-controlled flip-flops, or bistables as they are sometimes called, are used extensively in digital circuits. The basic building blocks of sequential circuits are the D (Delay) and the JK flip-flops. The D flip-flop simply delays a signal for one clock period. The JK flip-flop behaves like the cross-coupled NOR latch but permits the input combination (1,1). These, along with their state tables, are illustrated in Figure 2.5. Another common flip-flop, the T (Toggle) flip-flop, switches state in response to every active clock edge. A well-known theorem in sequential machine theory states that any of these circuits can be configured to emulate any of the others. For example, if the *J* and *K* inputs to a JK flip-flop are both tied to logic 1, the resulting circuit becomes a T flip-flop. Note that the $\overline{Preset}$ and $\overline{Clear}$ inputs on the D and JK flip-flop of Figure 2.5 are active low, so a logic 0 on the $\overline{Preset}$ input forces

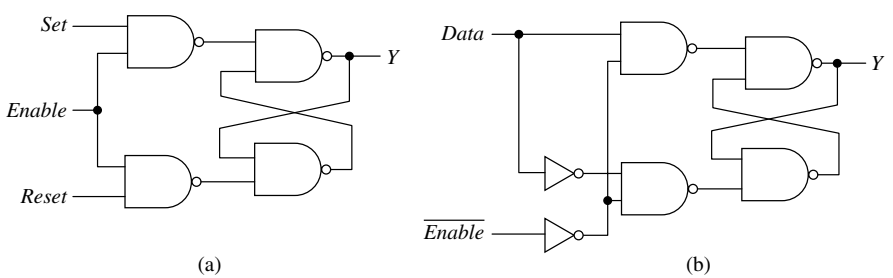


Figure 2.4 Gated latches.

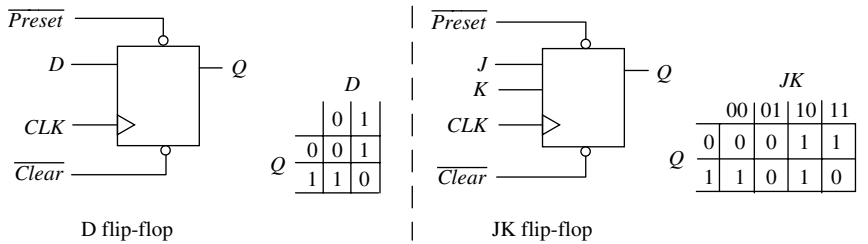


Figure 2.5 The standard flip-flops.

the Q output of these flip-flops to switch to a logic 1, while a 0 on the $\overline{Clear}$ forces Q to a logic 0. The clock input (CLK) is active on a positive edge for both the D and JK flip-flops.

The latch is similar in behavior to the D flip-flop. However, it is level-sensitive rather than edge-sensitive, meaning that the clock is replaced by an enable (EN) input and the value at the Data input appears at the output whenever the EN input is active. When EN switches to the inactive state, the value at the Q output is unaffected by signal changes at the Data input. Like the $\overline{Preset}$ and $\overline{Clear}$ lines, an active low Enable is represented by a bubble at the EN input.

The flip-flops depicted above can be implemented as *level-sensitive flip-flops* or as *edge triggered flip-flops*. A level-sensitive flip-flop responds to a high or low clock level, whereas an edge-triggered flip-flop responds to a rising or falling clock edge. The flip-flop in Figure 2.6 is a level-sensitive JK flip-flop implemented in a master/slave configuration. When the clock is high, data can enter the first stage or master. When the clock goes low, the data in the first stage are latched and the second stage, the slave latch, becomes transparent so data that was in the first stage are now transferred to the outputs.

The edge-triggered D flip-flop (DFF), shown in Figure 2.7, is somewhat more complex in its operation.[10] It has $\overline{Preset}$ and $\overline{Clear}$ lines with which the output Q can be forced to either a 1 or 0 state independent of the values on the *Data* and *Clock* lines. When the $\overline{Preset}$ and $\overline{Clear}$ are at 1 and the clock is low, then the complement of the value at the *Data* input appears at the output of N_4. Also, under these conditions, the output of N_1 has the same value as the *Data* input. Therefore, the input to N_2 at this time matches the value on the *Data* line, and the value on the input to N_3 is the complement of the value on the *Data* input.

When *Clock* goes high, the values at the inputs to N_2 and N_3 appear, inverted, at their outputs. They are then inverted once again as they go through N_5 and N_6 so that the output of N_5 matches the value on the *Data* line. There is an important point to note about this configuration: If *Data* is low when *Clock* goes high, then the output of N_3 goes low and prevents further changes in *Data* from propagating through N_4. If *Data* is high, then when *Clock* goes high, the high value at the output of N_1 causes a 0 to appear at the output of N_2. The 0 blocks changes at the *Data* input from propagating through N_1 and N_3.

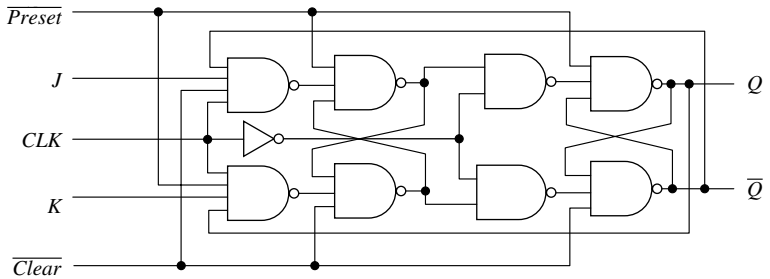


Figure 2.6 Level-sensitive JK flip-flop.

The circuit is sensitive to the rising edge of the *Clock* input. Data cannot get through N_2 and N_3 when *Clock* is low, and shortly after *Clock* goes high the data are latched so the flip-flop is insensitive to further changes at the *Data* input. However, data changes *during* the positive edge transition can cause unpredictable results. Therefore, these flip-flops are usually specified by their manufacturers with two key parameters: setup and hold time. S*etup time* is the interval during which a signal must be stable at an input terminal prior to the occurrence of an active transition at another input terminal. *Hold time* is the interval during which a signal must be stable at an input terminal following an active transition at another input terminal. In the flip-flop of Figure 2.7, setup and hold specify the duration of time during which the *Data* input must be stable relative to the *Clock* input.

With several levels of abstraction available for representing circuit behavior, it is reasonable to ask, "At what level of abstraction should a circuit be described?" There is no clear-cut answer to this question. Different engineers, with different objectives, find it necessary to work at different levels of abstraction. Consider the following example:

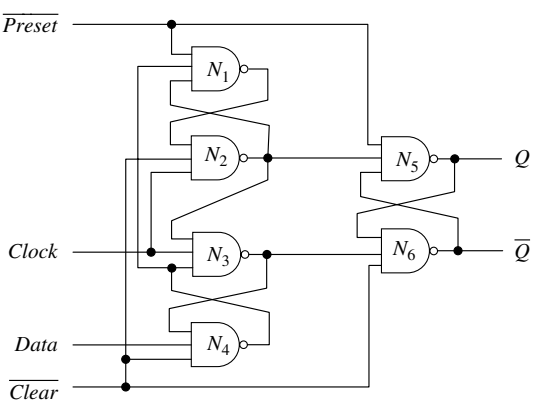


Figure 2.7 Edge-triggered delay flip-flop.

Example The frequency divider in Figure 2.8(a) may appear to be well-behaved. But if the latches are designed and used as shown in Figure 2.8(b), a pulse can be seen that the designer may not have anticipated.[11] If the unwanted pulse contains enough energy, the following flip-flop may be clocked more often than expected. ■■

Engineers responsible for designing and characterizing circuits for cell libraries must be aware of, and must document, precise details of a circuit's operation. Logic designers who instantiate that circuit in their design must be aware that the *Enable* has a minimum pulse width requirement of 8 ns.

2.6 THE COMPILED SIMULATOR

Compilers for programming languages can be characterized as compiled or interpreted. Simulators are similarly characterized as compiled or event-driven. The *compiled simulator* is created by converting a netlist directly into a series of machine language instructions that reflect the functions and interconnections of the individual elements of the circuit. For each logic element there exists a series of one or more machine language instructions and a corresponding entry in a circuit value table that holds the current value for that element. The *event-driven simulator*, sometimes called *table-driven*, operates on a circuit description contained in a set of tables, without first converting the network into a machine language image. We will first examine the compiled simulator.

The compiled simulator is constructed using the host computer's repertoire of machine language instructions. Each element in the circuit is evaluated using one or more instructions of the host computer. The results are stored in a table that contains

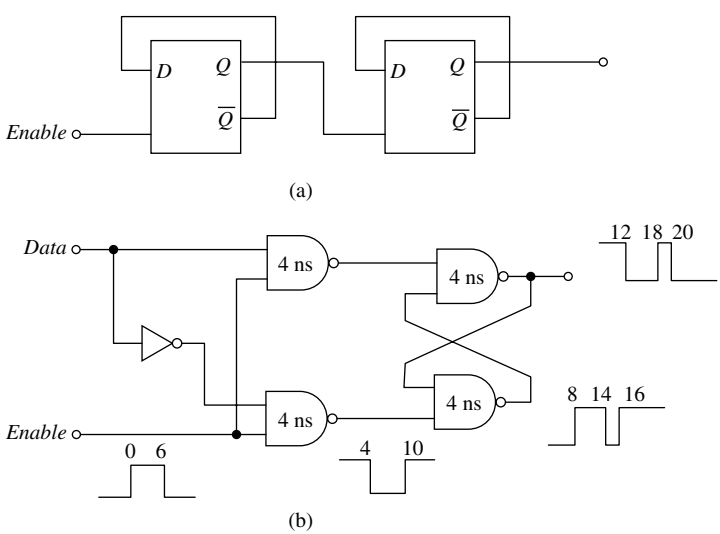


Figure 2.8 Frequency divider with spurious pulse.

an entry for each logic element being simulated. The instructions that simulate the circuit elements obtain their required input values from this table and store their results back into the table. Circuit preparation for simulation includes rank-ordering, defined below:

Definition 2.1 A *state point* is any primary input, primary output, or latch/flip-flop input or output. Primary inputs and latch/flip-flop outputs are called *input state points*. Primary outputs and latch/flip-flop inputs are called *output state points*.

Definition 2.2 A *cone*, also called a *cone of logic*, is the set of elements encountered during a backtrace from an internal circuit node, called the *apex*, to input state points.

Definition 2.3 A *predecessor* of a logic element is a logic element that lies in its cone.

Definition 2.4 A cone of logic is *rank-ordered*, sometimes said to be *levelized*, if the elements in the cone are numbered such that every element in the cone has a number that is greater than that of any of its predecessors.

Definition 2.5 The *level* of a logic element in a combinational circuit is a measure of its distance from the primary inputs. For any given gate, the level assigned is one greater than the highest level assigned to the gates that drive it. The level of the primary inputs may be chosen to be 0 (0-origin) or 1 (1-origin).

The apex of a cone often coincides with an output state point, but may be any internal node. When backtracing from an apex to input state points, all of the elements driving each element encountered during the backtrace are included in the cone of logic. The input state points are the drivers of the circuit defined by the cone. Note that if a cone is rank-ordered, then any sub-cone contained in that cone is also rank-ordered. The simulator takes advantage of rank-ordering to ensure that no element is evaluated until all of its predecessors have been evaluated. In Figure 2.9 the input to flip-flop *M* is an output state point. The cone of logic driving that state point, or apex, indicated by the dashed lines, contains the elements *G*, *H*, *I*, *J*, and *K*. The input state points that drive this cone are the primary inputs *B*, *C*, *D*, *E*, *F* and the output of flip-flop *A*.

A program for rank-ordering elements in a circuit begins by marking all of the primary inputs. Then, each unmarked element in the circuit is examined. It is marked if all of its inputs have been marked. If level numbers are required, the level assigned to each gate is the highest level among the driving gates, plus one. After all elements have been processed, if at least one additional element has been marked and if there are elements that have not yet been marked, the process is repeated. For a combinational circuit, the process terminates after a finite number of passes through the circuit. For a sequential circuit, elements in a loop may not get marked because they are interdependent; for example, element *A* cannot get marked because

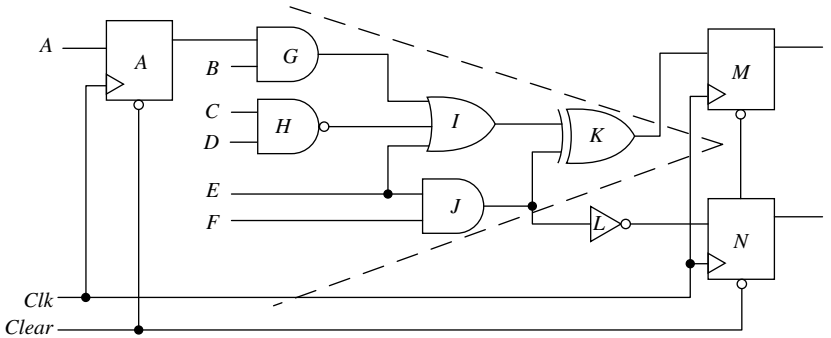


Figure 2.9 Circuit for simulation example.

element *B* has not been marked, and element *B* cannot get marked because element *A* has not been marked. A procedure for dealing with sequential loops is described in Section 5.3.2. Here we illustrate the operation of the compiled simulator.

Example A simulator will be created for the cone of combinational logic driving flip-flop *M* in Figure 2.9. It will use assembler language instructions for the 80×86 microprocessor.

```
; Set up stack for return values
PUSH   DS     ; Put return addr. on stack
MOV    AX,0   ; Clear register
PUSH   AX     ; Put return addr. (0) on stack
; Initialize data segment address
MOV    AX, DSEG           ; Initialize DS
MOV    DS, AX             ; -- by way of Reg. AX
; Begin simulation
MOV    AX, PI_TABLE       ; Load input A into Reg AX
MOV    BX, PI_TABLE + 2   ; Load input B into Reg BX
AND    AX, BX             ; G = A & B
MOV    GATE_TABLE, AX     ; Store result for gate G

MOV    AX, PI_TABLE + 4   ; Load input C into Reg AX
MOV    BX, PI_TABLE + 6   ; Load input D into Reg BX
AND    AX, BX             ; compute C & D
XOR    AX, OFFFFFH        ; Compute !(C & D)
MOV    GATE_TABLE + 2, AX ; H = !(C & D)

MOV    AX, PI_TABLE + 8   ; Load input E into Reg AX
MOV    BX, PI_TABLE + 10  ; Load input F into Reg BX
```

```
AND     AX, BX              ; Compute E & F
MOV     GATE_TABLE + 6, AX  ; J = E & F

MOV     AX, GATE_TABLE      ; Load value of G into AX
MOV     BX, GATE_TABLE + 2  ; Load value of H into BX
OR      AX, BX              ; compute G | H
MOV     BX, PI_TABLE + 8    ; Load input E into Reg BX
OR      AX, BX              ; Compute result, gate I
MOV     GATE_TABLE + 4, AX  ; Store result for gate I

MOV     AX, GATE_TABLE + 4  ; Load value of I into AX
MOV     BX, GATE_TABLE + 6  ; Load value of J into BX
XOR     AX, BX              ; Compute I ^ J
MOV     GATE_TABLE + 8, AX  ; Store K = I ^ J
RET
```

The network is compiled into machine code by a preprocessor that reads a description of the circuit expressed in terms of logic elements and interconnecting nets. A table called PI_TABLE contains an entry for each primary input, while another table, called GATE_TABLE, contains an entry for each gate in the circuit. There is a one-to-one correspondence between primary inputs and locations in PI_TABLE, and between circuit nets and locations in GATE_TABLE. The first step in this simulation is to load the locations represented by PI_TABLE into Reg. AX and PI_TABLE + 2 into Reg. BX. The values on the two primary inputs represented by these locations are ANDed together and the result stored in GATE_TABLE, at a location corresponding to the output of gate *G*. The next group of instructions compute the value on the NAND gate *H*. Note that the host machine's XOR instruction is used, together with the argument 0FFFFH, to complement the result before storing it at GATE_TABLE + 2.

The remaining gates are processed in similar fashion, and then the simulator returns to the calling program. Note that when simulating the exclusive-OR gate the simulator stores a result for gate *I* and then immediately loads the same value into Register AX. Since the simulator is called repetitively with many input vectors, every effort should be made to optimize its performance. This can be done by rank-ordering the circuit. If a gate drives another gate, all of whose other inputs have been processed, then the destination gate satisfies the rank-order criteria and can be the next gate simulated. In that case, the value in the accumulator can be used without being reloaded. It will still be necessary to save the calculated result in GATE_TABLE if the driving gate drives two or more destination gates, or if the control program must provide the ability to inspect intermediate simulation results on internal circuit nets after a simulation pass. ■■

The compiled simulator can also be implemented using two tables or arrays: the READ array and the WRITE array. In this implementation it is not absolutely

necessary to rank-order a circuit. As each vector is read, new values on primary inputs are stored in the READ array. Each element is then simulated as before, except that they may be processed in random order. When an element is simulated, its input values are obtained from the READ array and its result is stored in the WRITE array.

After all elements have been simulated, contents of the READ and WRITE arrays are compared. If they differ, the contents of the WRITE array are transferred to the READ array and the circuit is again simulated. [In practice, it is simpler to exchange names; the READ (WRITE) array in pass n becomes the WRITE (READ) array in pass $n + 1$.] Eventually, after a finite number of passes, contents of the two arrays must match if simulating a combinational circuit and the simulator can go on to the next input vector. Although this obviates the need for rank-ordering, it may be quite inefficient, requiring several passes before all input changes propagate to the outputs.

2.6.1 Ternary Simulation

In sequential circuits the values on many internal nets are determined by values on feedback lines. When power is first applied to a circuit, these values are indeterminate; they do not assume known values until the circuit is reset or until the latches and flip-flops are loaded with known values from other circuit elements on which they are functionally dependent. Hence it is necessary, at a minimum, to be able to represent a third value, the indeterminate state. This requires the use of two binary values to represent the three simulation values. One such mapping establishes the following correspondence between the three simulation values and the two-bit vectors:

0	0,0
1	1,1
X	0,1

The simulation program must be expanded accordingly, but first the operations on these two-bit vectors must be defined. It turns out that the processing is similar to processing of single-bit values in most cases. For example, to AND a pair of arguments, individual bit positions are ANDed. The OR operation behaves similarly. Primitives that invert arguments, such as the Inverter and the exclusive-OR, require special attention because a (1,0) is not the complement of an X. The inverter can be processed by complementing the individual bits and swapping them. The exclusive-OR of variables A and B is complicated by the fact that A and B could both be X. The computation may best be processed as $A \cdot \overline{B} + \overline{A} \cdot B$.

2.6.2 Sequential Circuit Simulation

When simulating a rank-ordered combinational circuit described in terms of standard logic gates, operation of the compiled simulator is quite straightforward. However, sequential logic requires additional processing before the compiled simulator

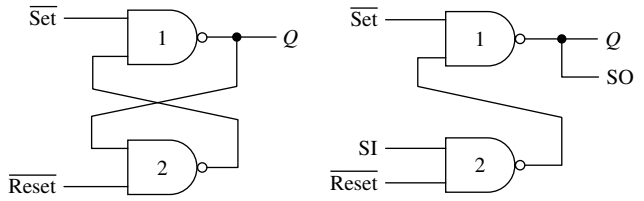


Figure 2.10 NAND latch.

can proceed. Consider the cross-coupled NAND latch of Figure 2.10(a). Before gate 1 is simulated, a value is needed from gate 2. But simulation of gate 2 requires a value from gate 1. The latch could be extracted in its entirety from the circuit and replaced with a call to an evaluation routine. Then, after simulation reached the point where all inputs to the latch were stable, the evaluation routine could determine the new values on the output of the latch. For a NAND latch the evaluation routine is not difficult to derive. For an asynchronous state machine comprised of many states, the task of creating an evaluation routine is formidable. An alternate approach is to cut feedback lines in the circuit model (cf. Section 5.3.2). If a cut is made from gate 1 to gate 2, the circuit model of Figure 2.10(b) is obtained.

After all loops in the circuit have been cut, the network is compiled. The circuit is now a pseudo-combinational circuit in which a feedback line has been replaced by a pseudo-input, designated SI, and a pseudo-output, designated SO. The pseudo-inputs are treated as primary inputs when rank-ordering and compiling the circuit.

Before simulation commences, the control program sets all pseudo-inputs to the X state. Then, during any single pass through the compiled simulator, each element is simulated once. It may be the case that the value on a pseudo-output is not the same as the value on the corresponding pseudo-input. In that case, the values on the pseudo-outputs are transferred to the corresponding pseudo-inputs and simulation is performed again. If the pseudo-outputs and pseudo-inputs continue to disagree, after some predetermined number of passes, it is concluded that the circuit is oscillating and the pseudo-inputs and pseudo-outputs that are oscillating are set to the X state. The control program then permits additional passes through the simulator, each time setting to X any additional pseudo-inputs that did not agree with their corresponding pseudo-outputs. Eventually the circuit stabilizes with some of the pseudo-inputs in the X state.

The pseudo inputs and pseudo outputs are analogous to having READ and WRITE arrays, but only for feedback lines. In fact, if the entire circuit is simulated using READ and WRITE arrays, then not only is it not necessary to rank-order the circuit, it is also not necessary to cut the loops. It is, however, still necessary to detect oscillations and inhibit them with the X state.

2.6.3 Timing Considerations

Elements used to fabricate digital logic circuits introduce delay. Ironically, although technologists constantly try to create faster circuits by reducing delay, sequential logic circuits could not function without delay; the circuits rely both on correct logical operation of the components in the circuit and on correct relative timing of signals passing through the circuit. However, this delay must be taken into account when designing and testing circuits. Suppose the inverter in the latch of Figure 2.8 has a delay of n nanoseconds. If *Data* makes a 0 to 1 transition and *Enable* makes a 1 to 0 transition approximately n nanoseconds later, the cross-coupled NAND latch sees an input of (0,0) for about n nanoseconds followed by an input of (1,1). This produces unpredictable results. The problem is caused by the delay in the inverter. A solution to this problem is to put a buffer in the noninverting signal path so that signals *Data* and $\overline{Data}$ reach the NANDs at the same time.

In the latch circuit just cited, a race exists. A *race* is a situation in which two or more signals are changing simultaneously in a circuit. The race may be caused by two or more input signals changing simultaneously, or it may be the result of a single input change propagating along two or more signal paths from a net with multiple fanout. Note that a latch or flip-flop implies a race condition since these devices will always have at least one element whose signal both goes outside of the device and also feeds back to an input of the latch or flip-flop. Races may or may not affect the behavior of a circuit. A *critical race* exists if the behavior of a circuit depends on the order in which signals arrive at a common function or device, such as a flip-flop. Such races can produce unexpected and unwanted results.

2.6.4 Hazards

Unanticipated events in circuits can result from logic conditions that have been ignored up to this point, namely, hazards. A *hazard* is a chance event; it is the possible occurrence in a circuit of a momentary value opposite to that which is expected. Hazards can exist in combinational or sequential circuits, and they can be the result of the way in which a circuit is designed or they may be an inherent property of a function. In sequential circuits it is possible for unwanted and unexpected pulses to occur in combinational logic and propagate to sequential elements where they can cause erroneous state transitions to occur. Consider the circuit of Figure 2.11. If $A = B = R = 1$ and S changes from 1 to 0, then by virtue of the delay associated with the inverter, both AND gates, and subsequently the OR gate, will have a 0 output for a period corresponding to the delay of the inverter. After that period, the output of the OR gate returns to 1, but the pulse may persist long enough to set the latch. That pulse, sometimes referred to as a *glitch* or *spike*, can be avoided by adding a third AND gate to create the product term $A \cdot B$. This term is added to the sum $S = A \cdot \overline{S} + S \cdot B + A \cdot B$, and the glitch is avoided.

The hazard just illustrated is called a *static hazard*. A static hazard exists if the initial and final values on a net are the same but at some intermediate time the net

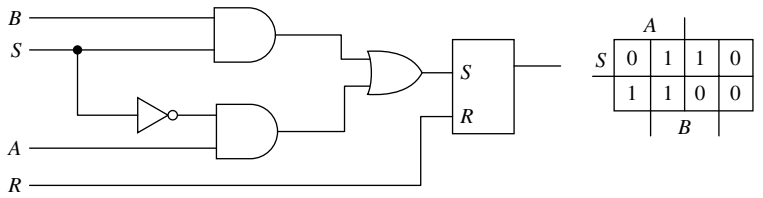


Figure 2.11 Circuit with hazard.

may assume the opposite value. If the initial and final values are 0 (1), then the hazard is sometimes called a *0-hazard* (*1-hazard*). A *dynamic hazard* exists if the initial and final values on a net are different and if, after achieving the final value, the net may assume the initial state one or more times. In other words, there is a dynamic hazard if it is possible to have $2n + 1$ transitions on a net for some integer n greater than 0. Note that the definition of a hazard only states that spurious transitions *may* occur; because of the variability of propagation delays, they may or may not actually occur.

Hazards are also categorized as logic or function hazards. Given a function f, a p-variable *logic hazard* exists for a p-variable input change U to V if

1. $f(U) = f(V)$.
2. All 2^p values specified for f in the subcube (cf. Section 4.3.1) defined by the p changing inputs are the same.
3. During the input change U to V a spurious hazard pulse may be present on the output.

The hazard illustrated in Figure 2.11 is a logic hazard. In the subcube defined by $A,S,B,R = (1,X,1,1)$, both values of f are 1. It has been shown that logic hazards can be eliminated by including all prime implicants in the implementation of a circuit.[12] A *function hazard* exists for the function f and the input change U to V iff*

1. $f(U) = f(V)$.
2. There exist both 1s and 0s specified for f within the 2^p cells of the subcube defined by the p inputs that changed.

Function hazards cannot be designed out of the circuit. Consider again the circuit of Figure 2.11. There is a function hazard when going from $A,S,B,R = (1,0,0,1)$ to $A,S,B,R = (0,1,1,1)$ because the input transition may go through the points $A,S,B,R = (0,0,0,1)$ and $A,S,B,R = (0,0,1,1)$ and the function f has value 0 at both points. The intermediate values assumed during operation will depend both on circuit delays and on the order in which the inputs change.

*We use iff as an abbreviation for "if and only if."

2.6.5 Hazard Detection

The compiled simulator performs logic evaluations. However, it ignores inherent delays in circuit elements. Furthermore, the cutting of feedback lines presumes that delay is lumped at that particular point where the cut occurred. Consider the NAND latch with the feedback line cut (Figure 2.10). If a transition occurs in which both $\overline{Set}$ and $\overline{Reset}$ lines change from 0 to 1, then the simulation result is totally dependent on where the cut occurred. With the cut illustrated in Figure 2.10(b), gate 2 will be simulated first and the latch will stabilize at $Q = 1$. If the cut was made from gate 2 to gate 1, then gate 1 will be simulated first and the latch will stabilize at $Q = 0$. This problem results from the assumption that the input changes arrived simultaneously and that the delays were lumped at one point. By moving the cut, in effect lumping the delay at another point in the circuit model, the simulator computed a different answer. In actual circuits, delay is distributed and the circuit could in fact oscillate if the input changes occurred sufficiently close together.

It was pointed out in Section 2.6.4 that circuit behavior can be affected by hazards. Hazards are a consequence of delay in circuit elements. The static hazard, which causes a momentary change to the opposite state on signal lines that should remain unchanged, may be of sufficient duration to cause a NAND latch to change state. If the inputs are $S,R = 1,1$ and the present state is $Q = 0$, then a momentary 1-0-1 glitch on the Set line could cause it to latch up in the $Q = 1$ state. But the compiled logic simulator will not detect glitches if it is only simulating logic 1 and 0.

To address this problem a ternary algebra, consisting of the symbols (0,1,X), was proposed.[12] The values were already in use to handle unknown values associated with feedback lines. However, ternary values can be applied to inputs whenever a change occurs. In effect, the ternary algebra describes the transition region in switching devices. It permits an approximation to continuous signals, as illustrated in Figure 2.12, by representing the "in between" time when a signal is neither a 0 or 1. In fact, if a signal fans out from a source, that signal could simultaneously represent a 0 to one device and a 1 to another device due to differences in switching characteristics of the driven devices. The ternary algebra tables for the AND gate and the OR gate are shown in Figure 2.13. The following two lemmas follow directly from the ternary algebra tables.

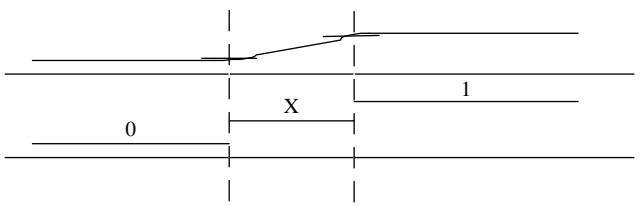


Figure 2.12 The transition region.

AND

	0	X	1
0	0	0	0
X	0	X	X
1	0	X	1

OR

	0	X	1
0	0	X	1
X	X	X	1
1	1	1	1

Figure 2.13 Ternary algebra tables.

Lemma 2.1 If one or more gate inputs are changed from 0 to X, or 1 to X, the gate output will either remain unchanged or change to X.

Lemma 2.2 If one or more gate inputs are changed from X to a known value, the gate output will either remain unchanged or change from X to a known value.

The following theorems flow from the lemmas:

Theorem 2.1 If one or more ternary inputs to a combinational logic network changes from 1 to X or 0 to X, then the network output either remains unchanged or changes to X.

Theorem 2.2 If one or more ternary inputs to a combinational logic network changes from X to 1 or X to 0, then the network output either remains unchanged or changes from X to 1 or X to 0.

Theorem 2.3 The output $f(a_1, ..., a_n)$ of a combinational logic network may change as a result of changing inputs $a_1, ..., a_n$ iff

$$f(\mathrm{X}, ..., \mathrm{X}, a_{p+1}, ..., a_n) = \mathrm{X}$$

With these theorems a pair of procedures can be defined for determining whether or not a circuit will be affected by static hazards, critical races, or essential hazards during a given input state change. Using the Huffman model, proceed as follows:

Procedure A. Determine all changing Y signals. Changing inputs are first set to X. If any Y_i outputs change to X, change the corresponding y_i inputs and resimulate. Continue until no additional Y_i changes are detected.

Procedure B. Determine which Y signals stabilize. Set changing inputs from X to their new binary state and simulate. If any Y_i changes from X to 1 or 0, then change the corresponding y_i and resimulate. Continue until no additional Y_i changes occur.

Theorem 2.4 If feedback line $Y_k = 1(0)$ after applying Procedure A and Procedure B to a sequential circuit for a given input-state change starting in a given internal state,

then the Y_k feedback signal must stabilize at 1(0) for this transition regardless of the values of the (finite) delays associated with the logic gates.

These theorems state that if ternary algebra is used when simulating, and unstable feedback lines are handled in accordance with procedures A and B, then:

1. Hazards, races and oscillations are automatically detected.
2. For a circuit with n feedback lines, at most $2n$ simulation passes are required.

Example For the NAND latch of Figure 2.10(b), the original input $\overline{Set} = \overline{Reset} = 0$ results in a 1 on pseudo-input *SI*. With ternary simulation the $\overline{Set}$ and $\overline{Reset}$ lines both switch from 0 to X, and then from X to 1. Procedure A is applied first. Gate 2 is simulated and the (1, X) combination on the inputs causes an X on the output. This value is input to gate 1 and, together with the X on the other input, causes gate 1 to switch to X. This X then appears on the pseudo-output.

Since the value on *SO* differs from the value on *SI*, the value on *SO* is transferred to *SI* and the circuit is resimulated with the X values on the $\overline{Set}$, $\overline{Reset}$ and pseudo-input. The circuit is now stable with an X on *SI* and *SO*. Procedure B is now applied. The inputs are changed to 1 and the circuit is resimulated. Note, however, that the X on the pseudo-input causes an X to occur on the output of gate 2; this in turn causes an X on the output of gate 1 and, subsequently, on the pseudo-output *SO*. The circuit is "stable" in the unknown state. ■■

2.7 EVENT-DRIVEN SIMULATION

A latch or flip-flop does not always respond to activity on its inputs. If an enable or clock is inactive, changes at the data inputs have no effect on the circuit. Compiled simulators in the past have used a method called *stimulus bypass* to take advantage of this fact.[13] Flip-flops were modeled as an integral body of machine code in which the first few instructions checked key inputs to determine if internal activity were possible. The property of digital networks, whereby a very small amount of activity occurs during a given time step, is often termed *latency*. As it turns out, the amount of activity within a circuit during any given timestep is often minimal and may terminate abruptly.

Since the amount of activity in a time step is minimal, why simulate the entire circuit? Why not simulate only the elements that experience signal changes at their inputs? This strategy, employed at a global level, rather than locally, as was the case with stimulus bypass, is supported in Verilog by means of the *sensitivity list*. The following Verilog module describes a three-bit state machine. The line beginning with "always" is a sensitivity list. The if-else block of code is evaluated only in response to a $1 \rightarrow 0$ transition (negedge) of the *reset* input, or a $0 \rightarrow 1$ transition (posedge) of the *clk* input. Results of the evaluation depend on the current value of *tag*, but activity on *tag*, by itself, is ignored.

```
module reg3bit(clk, reset, tag, reg3);
input clk, reset, tag;
output reg3;
reg [2:0] reg3;
always@(posedge clk or negedge reset)
  if(reset == 0)
    reg3 = 3'b110;
  else      // rising edge on clock
    case(reg3)
      3'b110:  reg3 = tag ? 3'b011 : 3'b001;
      3'b011:  reg3 = tag ? 3'b110 : 3'b001;
      3'b001:  reg3 = tag ? 3'b001 : 3'b011;
      default: reg3 = 3'b001;
    endcase
endmodule
```

Verilog will be used in this text to describe circuits. The reader not familiar with Verilog, but familiar with C programming, should be able to interpret the Verilog examples with little difficulty since Verilog is, syntactically, quite similar to C, and the examples in this text use only the most basic features of the language. The interested reader not familiar with HDLs should consult texts dedicated to Verilog[14] and VHDL.[15] The IEEE Verilog Language Reference Manual (LRM) is another valuable source of information.[16]

When a signal change occurs on a primary input or the output of a circuit element, an *event* is said to have occurred on the net driven by that primary input or element. When an event occurs on a net, all elements driven by that net are evaluated. If an event on a device input does not cause an event to appear on the device output, then simulation is terminated along that signal path.

Event-driven simulation can be performed in either a zero or a nominal delay environment. A *zero-delay* simulator ignores delay values within a logic element; it simply calculates the logic function performed by the element. A *nominal-delay* simulator assigns delay values to logic elements based on manufacturer's recommendations or measurements with precision instruments. Some simulators, trying to strike a balance between the two, perform a *unit-delay* simulation in which each logic element is assigned a fixed delay, and since the elements are all assigned the same delay, the value 1 (unit delay) is as good as any other.

The nominal delay simulator can give precise results but at a cost in CPU time. The zero delay simulator usually runs faster but does not indicate when events occur, so races and hazards can present problems. The unit-delay simulator lies between the other two in range of performance. It records time units during simulation, so it requires more computations than zero-delay simulation, but the mechanism for scheduling events is simpler than for time based simulation. However, regarding all element delays as being equal can produce inaccurate results in timing sensitive circuits and may give the user a false sense of security. Unit delay

simulation in sequential circuits does, however, have the advantage that time advances; so if oscillations occur, they will eventually reach the end of the clock period and be detected without a need for additional code dedicated to oscillation detection.

2.7.1 Zero-Delay Simulation

Event-driven, zero-delay simulation will be considered first. The zero delay is obviously not a delay at all; the term simply denotes a simulation environment in which propagation delay is ignored. When performing event-driven simulation, it is not necessary to rank-order the circuit. Before simulating the first input pattern, all nodes are initialized to X. Then, whenever an element assumes a new value, whether it be a primary input changing as a result of new stimuli being applied or an internal element changing as a result of event propagation, any elements driven by that element are simulated.

The Event-Driven, Zero-Delay Simulator An event-driven, zero-delay simulator can be implemented by means of the READ/WRITE arrays described earlier, and associating a flag bit with each entry in the arrays. If an event occurs at the output of an element, the elements affected by that event are identified and flagged for simulation in the next pass. When no new events occur during a pass, the circuit is stable. Alternatively, elements that must be simulated in the next pass can be placed on a first-in first-out (FIFO) stack, assuming they are not already on the stack. When the stack is empty at the end of a pass, the circuit is stable.

Example Event-driven simulation will be illustrated using the circuit in Figure 2.14. At the first time interval, denoted by column heading t_0, all elements driven by inputs 1, 2, 3, and 5 are simulated. Simulation causes the outputs of gates 6 and 7 to switch from X to 0. Simulation of gate 8 produces a 1 on its output. These changes cause gate 9 to be simulated, with the result that a 1 appears on its output. At time t_1, input 1 changes from a 1 to 0. However, there is no change on the output of gate 6, so simulation for time t_1 is done. Input 2 changes at time t_2, causing gate 9

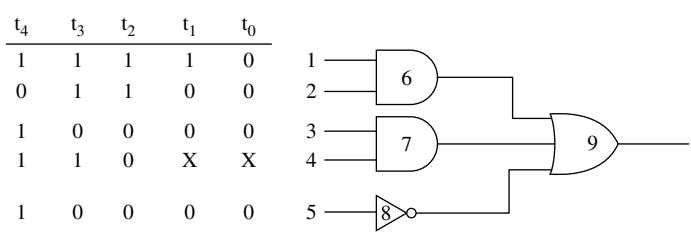


Figure 2.14 Zero-delay simulation.

to be simulated. The output of gate 9 does not change. Gate 7 is simulated at time t_3, but again no output activity occurs. At time t_4, input events cause all gates to be simulated. ■■

In this tiny example it is difficult to appreciate the value of event driven simulation, but in a circuit containing many thousands of gates, a situation as occurred in time t_1 can happen frequently and can provide substantial savings in computer time. The simulation at time t_1 was terminated almost immediately because a single input change occurred that had virtually no effect on the circuit.

Hazard Detection Using Multiple Values The three-valued hazard analysis can be used with event-driven, zero-delay simulation without having to rank-order or cut the feedback lines in the model. Simply perform an intermediate X value simulation on all changing inputs and the circuit will stabilize. However, the three-valued simulation will not detect dynamic hazards. A nine-valued simulation can be performed to detect dynamic hazards.[17] The nine values denote various combinations of stable and changing signals. The values are used in conjunction with operator tables for the basic logic operations. The symbols are defined in Table 2.1. The operation table for the AND gate is given in Table 2.2. From this table, any pair of incoming signals to a two-input AND gate can be processed to determine whether the result will cause a static or dynamic hazard. For example, if one of the inputs is a constant 0, the output must be a constant 0. With a static 0-0 hazard on one input, there will always be a static 0-0 hazard on the output unless another input to the AND gate blocks it with a constant 0. The circuit in Figure 2.15 illustrates creation of a dynamic 0-1 hazard in a pair of NAND gates. The table for the AND gate is easily extendable to n, $n \geq 2$, since the AND operation is commutative and associative.

Table 2.3 gives the hazard detection results for the NAND latch of Figure 2.10(a). In this table the columns correspond to values on the $\overline{Reset}$ input and the rows correspond to values on the $\overline{Set}$ input. The values in the lower right quadrant of this table contain two values. The actual value assumed at the output depends on the previous state of the latch. If the Q output is presently true, then the first value is assumed. If false, then the second value is assumed.

TABLE 2.1 Symbols for Hazard Detection

Symbol	Meaning	Complement
0	constant 0	1
1	constant 1	0
/	dynamic hazard 0-1	\
\	dynamic hazard 1-0	/
^	0-1 transition, hazard-free	v
v	1-0 transition, hazard-free	^
M	0-0 static hazard	W
W	1-1 static hazard	M
*	race condition	*

TABLE 2.2 Hazard Detection During and Operation

AND	0	1	M	W	*	^	∨	/	\
0	0	0	0	0	0	0	0	0	0
1	0	1	M	W	*	^	∨	/	\
M	0	M	M	M	M	M	M	M	M
W	0	W	M	W	*	/	\	/	\
*	0	*	M	*	*	*	*	*	*
^	0	^	M	/	*	^	M	/	M
∨	0	∨	M	\	*	M	∨	M	\
/	0	/	M	/	*	/	M	/	M
\	0	\	M	\	*	M	\	M	\

TABLE 2.3 Hazard Detection for NAND Latch

	0	^	M	/	*	1	∨	W	\
0	1	∨	W	\	*	0	^	M	/
^	1	*	W	*	*	0	^	*	/
M	1	∨	W	\	*	0	^	M	/
/	1	*	W	*	*	0	^	*	/
*	1	*	W	*	*	*	*	*	*
1	1	1	1	1	*	1-0	1-^	1-*	1-^
∨	1	∨	W	\	*	∨-0	W-^	\-M	W-/
W	1	*	W	*	*	∨-0	W-^	*	W-/
\	1	∨	W	\	*	∨-0	W-^	\-M	W-/

2.7.2 Unit-Delay Simulation

Unit-delay simulation operates on the assumption that all elements in a circuit possess identical delay time. It has the advantage that it is easier to implement than nominal-delay simulation. In fact, when every element has unit delay, the READ/WRITE array implementation described in Section 2.6 for zero delay simulation is sufficient since each pass through the simulator corresponds to advancement of events through one level of logic. Primary inputs can switch values while other events are still propagating to outputs. When copying the WRITE array into the READ array, if entries that change during the simulation pass are flagged, then performance can be enhanced by simulating only those elements that experience events at their inputs.

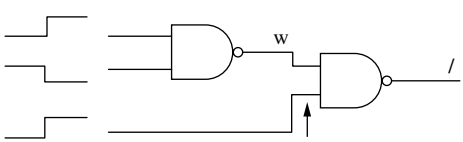


Figure 2.15 Creation of dynamic hazard.

When creating test stimuli for a timing-sensitive circuit, the unit-delay simulator can give a false sense of security. Timing for the actual circuit may not resemble the results predicted by the unit-delay simulator. When simulating test stimuli in order to generate a test program, it may be necessary to insert additional gates with unit delay into the circuit model so as to force the simulator to predict correct circuit response for a given set of input stimuli. Another drawback to unit-delay simulation is the fact that, because all elements have nonzero delay, the circuit cannot be rank-ordered for simulation purposes. Hence, elements may be unnecessarily evaluated several times in a single period.

Unit delay can be useful in applications such as gate arrays. These are integrated circuits made up of a fixed array of rows and columns. At the intersection of each row and column is an identical device that may be a NAND gate, a NOR gate, or a collection of transistors and resistors. The logic designer implements a function on a gate array by specifying the connections of switching elements at row/column intersections. Metal layers are provided to accomplish the interconnections. Switching elements connected in this way often have the same switching speed, in which case a unit delay is meaningful. If the switching speeds are integral multiples of one another, then unit delay can still be effectively employed.

2.7.3 Nominal-Delay Simulation

Zero-delay simulation with three or nine values can provide correct simulation results because it can accurately predict hazards and races. However, it is worst-case or pessimistic because it ignores the time dimension and collapses all computations into zero time. As a result, it may see conflicts that do not occur in real time. A designer may intend for an asynchronous state machine to receive two or more events during the same clock period. The designer will make use of the delay in the devices and, if necessary, incorporate additional delay into signal paths to ensure that the signals arrive at the state machine in the correct sequence. The zero-delay simulator, not recognizing the delay information, concludes that a race exists and that an unpredictable state transition will occur. As a result, it may put the state machine into an indeterminate state.

Nominal delay represents the real delay of a device. However, the accuracy of that representation depends on how accurately the delay is calculated. For example, the nominal delay along a signal path may be calculated solely from delay values given for individual cells residing in a macrocell library. There was a time in the past when these values would have been sufficient to give reasonably accurate delay values. Now, however, for devices operating on the leading edge of technology, the contribution to total circuit delay by the components may be exceeded by the delay inherent in their interconnections. As a result, an accurate accounting of the total delay between points in a circuit is often possible only after layout, when delays are calculated for the components and interconnections, and back-annotated to the circuit model.

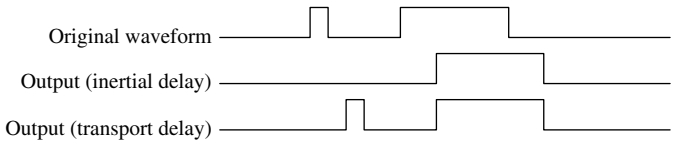


Figure 2.16 Transport versus inertial delay.

A number of types of delays exist for describing circuit behavior. The two major hardware description languages, Verilog and VHDL, support *inertial delay* and *transport delay.* Inertial delay is a measure of the elapsed time during which a signal must persist at an input of a device in order for a change to appear at an output. A pulse of duration less than the inertial delay does not contain enough energy to cause the device to switch. This is illustrated in Figure 2.16 where the original waveform contains a short pulse that does not show up at the output. Transport delay is meaningful with respect to devices that are modeled as ideal conductors; that is, they may be modeled as having no resistance. In that case the waveform at the output is delayed but otherwise matches the waveform at the input. Transport delay can also be useful when modeling behavioral elements where the delay from input to output is of interest, but there is no visibility into the behavior of delays internal to the device.

The length of time required to propagate a signal from one physical point to another through wire is sometimes referred to as *media delay*; this time is approximately one nanosecond per foot of wire. As circuits continue to shrink and devices continue to switch at faster speeds, the media delay becomes a significantly larger percentage of the total elapsed time in a circuit and it is not unusual for media delay to account for a majority of the cycle time on a high-performance circuit.

The amount of time it takes to switch from 0 to 1 is called *rise time*. The delay in a transition from 1 to 0 is called *fall time.* The elapsed time required to switch from a 1 or 0 to Z is called *turn off delay.* Delays can also be characterized according to whether they represent minimum delay, typical delay, or maximum delays. Thus the Verilog tranif1 primitive could have as many as nine delay values associated with it. These include min, typical, and max for each of the rise, fall, and turn-off delays. Differences in rise and fall times are often due to capacitance and storage effects of transistors used to implement switching circuits, whereas differences in minimum, typical, and maximum delay values are more likely to result from variations during manufacturing.

Manufacturer's data books identify several kinds of propagation delay, and the list of delays will generally depend on the product. For example, the manufacturer's data book may specify t_{DOV} (Data Out Valid) to be the interval from when an active clock edge appears at a device to when an n-wide output data bus contains valid data for that device. A complete characterization of a complex functional unit usually contains many such time intervals.

Ambiguity delay is sometimes used to express the difference between nominal and maximum or minimum delays. This may be of use in PCBs populated by many

ICs—some of which may run faster than nominal, and others of which may run slower than nominal. This ambiguity may have to be considered if behavior of a PCB does not match simulation predictions.

When applying a test to a circuit on a tester, ambiguity delay can result from skew at the tester pins. Although the test program may specify that two or more signals change at the same time, the actual time between events on the tester may occur picoseconds or nanoseconds apart due to various physical considerations. In asynchronous circuits, in particular, it may be necessary to use the simulator to determine if this skew or ambiguity delay represents a problem. This can be done by inserting random delays at the circuit inputs so that events no longer occur simultaneously at the start of a tester cycle. If the circuit is sensitive to delays at the inputs, staggering the switching times may reveal the problems.

2.8 MULTIPLE-VALUED SIMULATION

When a device first powers up, there is uncertainty as to the states of its storage elements—for example, flip-flops and latches. Races, hazards, undefined inputs, and transition regions (when a signal value is between a 0 and 1) are additional factors that contribute to uncertainty. Ternary simulation, which adds the symbol X to the binary {0,1} values, has been used to represent indeterminate values. It is also useful for resolving values in designs where two or more circuits may simultaneously drive a bus, although, as we shall see, conflicts can sometime be resolved by examining combinations of signals. The resolution of these combinations is not always performed in accordance with the rules of Boolean algebra. The evaluation of transistor-level circuits also depends on multiple values, as well as signal strengths.

A tri-state device is one in which the output may assume a logic 1 or logic 0 state, or the output may be disconnected from the remainder of the circuit, in which case the device has no effect on the circuit. In this third state, the output is in a high-impedance state. This circuit is used when the outputs of two or more devices are tied together and alternately drive a common electrical point, called a bus. A circuit employing two tri-state drivers is illustrated in Figure 2.17.

When input $A = 1$, the tri-state device controlled by A behaves as an ordinary buffer. When $A = 0$ the output E_1 assumes the high impedance state, represented by the symbol Z. With a high impedance capability, two or more tri-state outputs can be

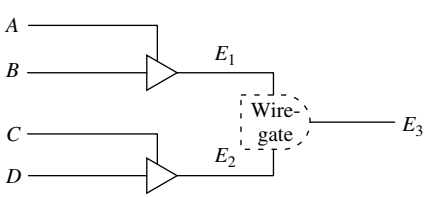


Figure 2.17 Circuit employing tri-state drivers.

tied directly together. However, if this is done, one rule must be observed. Two tri-state controls must not be active at the same time. In Figure 2.17, *A* and *C* must not be simultaneously high. If both are high and if the output of one device is low and the output of the other is high, then there is a low-resistance path from power to ground; in a very short time, one or both of the devices could overheat and become permanently damaged.

Note that the wire-gate in Figure 2.17 is represented by a *resolution function*, its purpose is to indicate to the simulator that there are two or more elements driving the net. A simulator could be designed to check every net for multiple drivers each time it computes the value at that net, but wire logic is more efficient: It is inserted into the circuit model when the model is created. Then, when the simulator encounters a wire-gate, it immediately enters a function that checks the outputs of all drivers and resolves the signal driving that net.

Although circuit designs normally do not permit two or more tri-state devices to be active simultaneously, design errors do occur and a logic designer may want to employ simulation in order to identify conditions wherein two or more drivers become simultaneously active. This requires that the simulator be able to correctly predict the behavior of bus-oriented circuits. It may be the case that, in the environment in which the IC is intended to operate, no pair of tri-state controls will be simultaneously active. But, when the IC is being tested, the tester represents an artificial environment. In this environment it is possible for signals to simultaneously activate two or more tri-state drivers. It is important that this situation be identified and corrected.

To resolve problems that may occur when the outputs of tri-state drivers are connected together, a set of simulation values incorporating both value and strength can be used. Figure 2.18 represents a resolution function, variations of which have been used in commercial simulation products. The values shown in Figure 2.18 are based on the binary values 0 and 1, but each of these values is extended by attaching strengths and then by adding ranges of signals. First consider the strengths. A logic 1 or 0 can be represented as strong, weak, or floating. The strong value is generated by a logic device that is driving an output. For example, an AND gate normally produces a driving 1 or 0 on its output. A weak value drives a node, but it has a weaker strength than the strong value. The weak signal could be produced by a small transistor. The floating 1 or 0 represents a charge trapped at a node. Ranges of

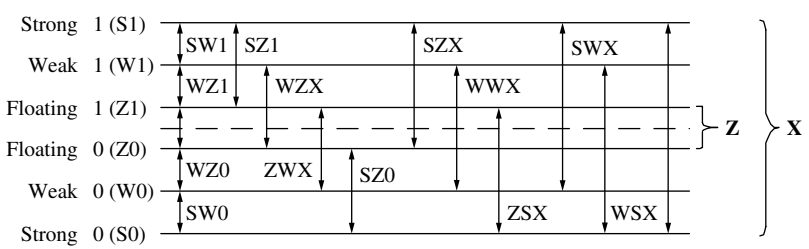


Figure 2.18 Logic ranges.

values occur when there is uncertainty as to the correct value. For example, if a tri-state device with an active high enable has a 1 on its input, and its enable has an X, the output of the device could be a strong 1 if the enable were a 1 or it could be a floating 1 if the enable were a 0.

Another ambiguous region occurs when a tri-state device with active-high enable has an X on its input and a 0 on its enable. The output in that case could be a floating 1 or a floating 0. The range Z1 to Z0 is represented as Z. To represent regions of ambiguity, the chart in Figure 2.18 extends the six initial value/strength entries by considering contiguous regions of values. The region from strong 1 to floating 1 is designated SZ1. The region from strong 1 to weak 1 is denoted SW1. The region from floating 1 to floating 0 is the familiar Z. If a signal is totally ambiguous (i.e., it could take on any of the six primary values), its value is totally unknown, or X. Other ranges may straddle both logic 1 and 0 values. For example, the value SZX straddles the range from a strong 1 to a floating 0; hence the third character in the identifier is an X. When the range lies completely in the region of logic 1 or logic 0, the third character is a 1 or 0.

Example To understand how the 21-value logic system can help to eliminate pessimism, consider again the circuit in Figure 2.17. Assume A = X and $B = C = D = 1$. If the circuit is simulated using ternary simulation, then the X at input A will produce an X at E_1. The signal at E_2 will be a 1. Since E_1 could be a 0 or 1, the wire-gate must be assigned the value X.

Now, suppose the circuit is evaluated using the 21-value system. With an X on the control input A and 1 on B, the value at E_1 could be a 1 or it could be a floating 1, denoted as Z1. With a 1 on E_2, a 1 on E_1 will resolve to a 1 at E_3. If E_1 has the value Z1, then the values 1 and Z1 at the wire-gate will again resolve to a 1 at E_3. In either case, the output is resolved to a known value. ■■

The 21-value system can be extended further. The value X is normally used to denote an unknown value. In Figure 2.17, if $E_1 = 0$ and $E_2 = 1$, the 21-value logic would assign an X to F_3. But, the consequences of these assignments are more than simply that the output is unknown. There is clearly a conflict, and it could cause permanent damage to an IC. Where two values are obviously and clearly driving a node to opposite values, this should be spelled out as a conflict. Thus a 22nd value, C, can be introduced, denoting a situation in which two devices are driving a node to opposite values. Another useful value is U (uninitialized). It can be assigned to all nodes at the start of simulation, and it can be used to identify nodes that have never been initialized during a simulation. If the signal U persists at a node to the end of simulation, the user can conclude that the node was never assigned a value. This may suggest that the node requires a reset capability.

The example in Figure 2.17 illustrates a situation where two devices whose outputs are connected together must not have conflicting values. In other situations it is not only permissible but desirable to have two or more devices simultaneously driving a net with conflicting values. This is the case in Figure 2.19. If the dynamic RAM (DRAM) cell is selected, by virtue of the word line WL being active, the bit line BL

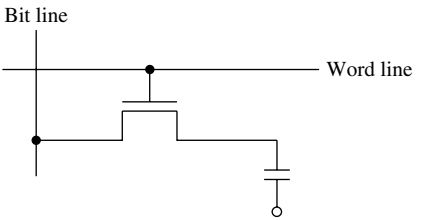


Figure 2.19 DRAM cell using transmission gate.

may be attempting to read the contents of the DRAM cell, or it may be trying to write a value into the cell. When writing into the cell, the value on the bit line is a strong 1 or 0, whereas the value in the capacitor is a floating 1 or 0. As a result, simulation of the circuit will result in a new value being written into the cell, regardless of what value had previously been there.

2.9 IMPLEMENTING THE NOMINAL-DELAY SIMULATOR

A number of factors must be taken into consideration when implementing a simulator. Events must be scheduled in the proper order in order to support concurrent operation of the elements in the circuit being simulated. Sometimes events that were scheduled have to be un-scheduled. Data structures and evaluation techniques must be defined. The choice of evaluation technique can have a significant impact on simulation performance. Other aspects of simulation must be decided. What kind of error handling is to be implemented for races, conflicts, setup and hold violations, and so on?

2.9.1 The Scheduler

Nominal-delay simulation recognizes the inherent delay in logic elements. However, because of this variability in their delays, individual elements cannot simply be placed in a FIFO queue as they are encountered. The element being simulated may experience an event at its output that occurs earlier than some elements previously scheduled and later than others. Hence, it must be scheduled for processing at the right time relative to other events. This can be done through the use of a linear linked list. In this structure an *event notice* is used to describe an activity that must be performed and the time at which it must be performed. The notices are arranged in the order in which they must be performed. Included in each event is a pointer to the next event notice in the list. When an event is to be scheduled, it is first necessary to find its proper chronological position in the linked list. Then, the pointer in the preceding link is made to point to the newly inserted event, and the pointer that was in the preceding event is inserted into this newly inserted event so that it now points to the next event.

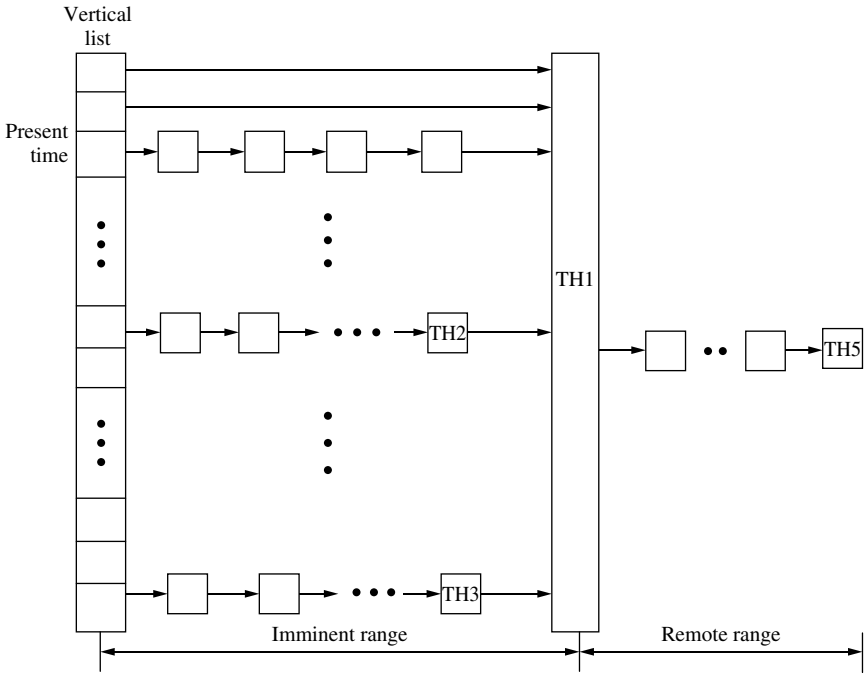


Figure 2.20 The converging lists scheduler.

To insert an event in this linked list, it is necessary to search, on average, half the elements in the linked list and modify two pointers. As the number of events grows, due to increased system size or increased activity, the average search time grows. To reduce this time, the scheduling mechanism shown in Figure 2.20 is used.[18] It is a combination of a vertical time mapping table, also called a delta-t loop or "timing wheel,"[19] and a number of horizontal lists. The vertical list represents integral time slots at which various events occur. If an event is to occur at time i, then either it is the first event, in which case a pointer is inserted at slot i to identify the event to be processed, or other events may already have been scheduled, in which case the present event is appended to the end of the list. Note that the event may be the result of a gate simulation, in which case the event is to be processed at future time, or the event may be a print request or other such request for service. These service requests scheduled on the wheel are often referred to as *bulletins*.

A further refinement, called nonintegral event timing,[20] defines the slots in the vertical list as intervals. If an event occurs within the time interval represented by that slot, then it must be inserted into its correct position in the horizontal linked list. Therefore, the search through a linked list must again be performed. However, the search is through a much smaller list. Performance is enhanced by making the vertical list as large as is practical, although not so big that a large average number of slots go unused.

To handle events that occur far in the future, imminent and remote ranges are used. These are implemented by means of thresholds shown in the converging lists scheduler of Figure 2.20. All but two of the wheel slots link directly to threshold TH1. The remaining two slots link first to TH2 and TH3, and then to TH1. From TH1, the linked list terminates on TH5, which represents infinity. The thresholds are control notices; they can be scheduled like elements and represent requests for service, such as printout of simulation results. When inserting an item into a horizontal list, if TH1 is encountered, then the item is inserted between TH1 and the item previously linked to TH1. If time of occurrence of an event exceeds imminent time, then it is inserted into its appropriate slot in the remote list. During simulation, if TH2 or TH3 is encountered, then imminent time is increased, the new maximum imminent time is stored in control notice TH1, and items from the remote range are retrieved and inserted (converged) into their proper place in the imminent range.

In order to obtain correct simulation results when an event is simulated, it is necessary that any change at the inputs cause a simulation using the values that exist on the other inputs at the time when the event arrives at the given input. Therefore, the input change is simulated immediately, but the output value is not altered until some future time determined by the delay of the element. This imitates the behavior of a logic element with finite, nonzero delay. An event appears at a gate input; and at some future time, depending on element delay, the effects (if any) of that event appear on the element output and propagate forward to the inputs of gates that are driven by that gate.

If simulation does not result in a change on the output of an element, it is tempting to assume that nothing further need be done with that element. However, it is possible that a simulation indicates no change, but a previously scheduled change occurred and presently exists on the scheduler. For example, suppose a two-input AND gate with propagation delay of 10 ns has values (1, 0) on its inputs at time t when a positive pulse of duration 3 ns reaches the second input. The simulation result at time t is a 1, which differs from the 0 presently on the output, so the event is placed on the scheduler for processing at time $t + 10$. At $t + 3$, when simulating the change to 0, the simulator computes a 0 on the output, which matches the present value. Therefore, the simulator may incorrectly conclude that no scheduling is required.

One solution to this problem is to always put the event on the scheduler regardless of whether or not there is a change on the output. Then, when it is processed later, if its output value is equal to its present value, drop it from further processing. In the example just given, the AND gate is simulated at time t and placed on the scheduler. It is simulated at $t + 3$ and again placed on the scheduler. At $t + 10$ it is retrieved from the scheduler and its output is checked. The current value is 0 and the new value is 1, so the element output is updated in the descriptor cell and the result propagated forward. At $t + 13$ the process is repeated, this time with the present value equal to 1 and the computed value switching back to 0.

Another approach that can save scheduling time makes use of a schedule marker. It is used as follows: Simulate the input event.

- If there is an output event, schedule the change and increment the schedule marker.

- If there is no output event and schedule marker equals 0, no activity is required.
- If there is no output event and schedule marker is greater than 0, schedule the change and increment the marker.
- When an output event is processed, decrement the schedule marker.

Occasionally an event on the output of an element is followed almost immediately by another event with a pulse duration less than the inertial delay of the element. In that case, the user may want to retain the glitch and propagate it to succeeding logic to determine if it could cause a problem. While the glitch should not propagate if all elements have delay values exactly equal to their nominal values, delay values that vary slightly from nominal can cause the pulse to exceed the inertial delay of the element.

It may be the case that the glitches are in data paths where, even if they do occur, they are not likely to cause any problems and their presence clutters up the output from the simulator. In that case it is desirable to suppress their effects. Consider a 2-input AND gate with t_p-nanosecond propagation delay and suppose its present input values are (1, 0). If it has inertial delay of t_i nanoseconds and if a pulse of duration t_g, $t_g < t_i$, appears on its lower input, then it is scheduled for a change at $t + t_p$ and again at $t + t_p + t_g$ nanoseconds. In that case, it would be desirable to delete the change at $t + t_p$ from the scheduler before it is processed since it would otherwise cause unwanted changes to be scheduled in successor elements.

If the time at which an element is placed on the scheduler is recorded, that information can be used to determine if the duration of the output signal value exceeds the inertial delay. In the situation just described, the time $t + t_p$ is recorded. When the next output change occurs at $t + t_p + t_g$, its time of occurrence is compared with the previous time. If the signal duration does not exceed the inertial delay, the recorded time of the previous change is used to search the appropriate linked list on the schedule for the event to be deleted. If a previous change occurred but its time was not recorded, it would be necessary to search all time slots on the scheduler between $t + t_p$ and $t + t_p + t_g$.

2.9.2 The Descriptor Cell

During simulation, information describing each element in a circuit is stored in a descriptor cell. The cell contains permanent information, including pointers for each input and output, and descriptive information about the element represented by that cell, such as its function and delay values. It also contains data that change during simulation, including the schedule marker and logic values on the inputs and outputs of the element. A descriptor cell is illustrated in Figure 2.21(a) for an element with one output. The first few entries point to devices that drive the inputs of the element represented by this descriptor cell. There is an entry for each input, and each entry has a field that indicates the element input number. Since input values are stored in the descriptor cell, the input number is used to access and update the correct bits in the descriptor cell during simulation. The last entry points to destination input(s) that are driven by this element.

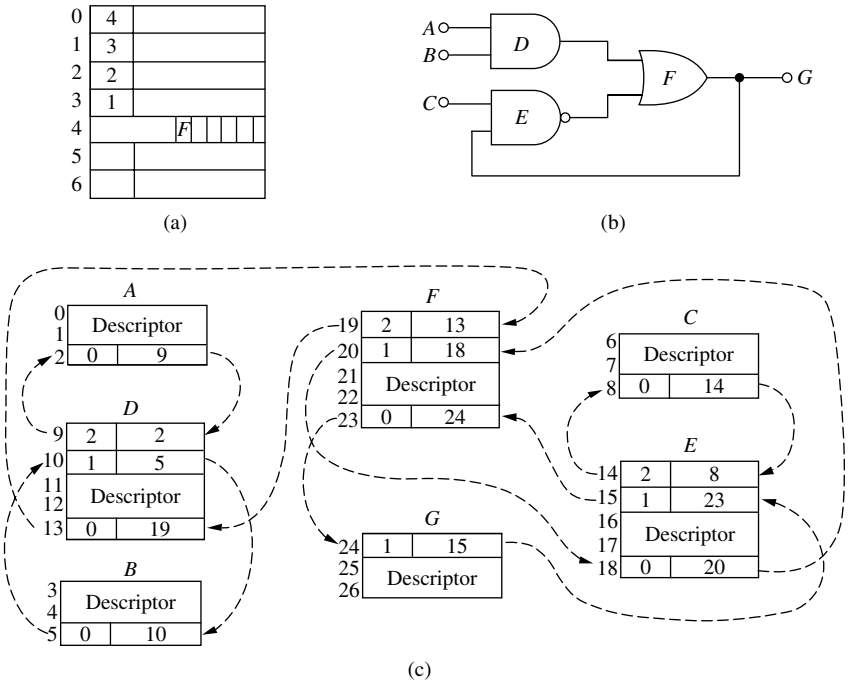


Figure 2.21 The descriptor cell.

An element with two or more outputs will have a corresponding number of output entries in the descriptor cell. A simple circuit and its descriptor cell model are illustrated in Figures 2.21(b) and 2.21(c), respectively. Each descriptor cell corresponds to an element in the circuit model, and the nets that interconnect circuit elements are represented in the model by linked lists that thread their way through the descriptor cells. For example, primary input *A* drives input 1 of gate *D*, which is located at memory location 9 in this example. Therefore, the output pointer of descriptor cell *A* points to location 9, corresponding to the first entry of *D*. Gate *F* fans out to two places so the linked list extends through the descriptor cell for *G*, and then to the descriptor cell for *E*. A pointer then returns to *F*, where the high order field is 0. In the configuration illustrated here, when traversing the linked list to find the fanout elements for a particular device, the traversal is halted when a word is encountered in which the high-order field is 0.

To illustrate the scheduling process using the scheduler and descriptor cells, suppose we want to schedule input *A* for a change at time t_i. To do so, we check the schedule marker *A*. If it is not busy, we take the output pointer from cell *A*, location 2, and attach it to the linked list at scheduler slot t_i (assumes an integral timing scheduler). If nothing is scheduled at time t_i, then schedule location t_i contained a pointer to one of the thresholds TH1, TH2, or TH3. The threshold pointer is placed in location 2, while schedule location t_i receives the value 9.

If other elements are already scheduled for time t_i, then this operation automatically links the descriptor cells. Suppose C had already been scheduled. Then the schedule contains the value 14 and location 8 contains the threshold pointer. To schedule a change on A, its output pointer is exchanged with the slot on the vertical list. Slot t_i on the vertical list then contains 9, location 2 contains the value 14, and location 8 contains a pointer to threshold TH1, TH2, or TH3. Therefore, at time t_i a change on the first input of D will be simulated, as will a change on E. When processing for time t_i is complete, all pointers are restored to their original values.

If an element is busy, as indicated by its schedule marker, and it must be scheduled a second time, it becomes necessary to obtain an unallocated memory cell for scheduling this second event. The address of the spare cell is placed in the schedule, and the spare cell contains a pointer to the cell to be scheduled. If other events are scheduled in the time slot, then this spare cell must also contain a link to the additional events.

Example The circuit in Figure 2.22 will be used to illustrate nominal delay simulation. Alphabetic characters inside the logic symbols represent gate names and the numbers represent gate delays. All nets are initially set to X. Detailed computations are shown in Table 2.4. At time t_0, input D changes from X to 0. At time t_2 the inputs are set to the values (X,1,1,1). At time t_4, input A changes from X to 0 and input C changes from 1 to 0. At t_8, input C changes back to 1. In this table, the times at which activity take place are indicated, as well as the values on the inputs and the gates at those times. For each of the logic gates, there are two values: The first is the logic value on the output of the gate, and the second is the value of the schedule marker. The comments indicate what activity is occurring. For example, at time t_0, input D changes, so gate F is simulated; its output changes from X to 0, so it is scheduled for time t_5 and its schedule marker is incremented to 1.

At time t_2, E and F are both simulated because of input changes. There is no change on the output of E and its schedule marker is 0, so it is not scheduled. However, F does change from its present value so it is scheduled for update at time t_7 and its schedule marker is again incremented. The remaining entries are similarly interpreted. Note that at time t_8 the output of F has the value 1, and it is simulated with (1,1) on its inputs. Although the simulation result is a 1, F is put on the scheduler because its schedule marker is nonzero. ■■

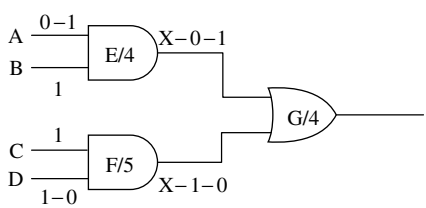


Figure 2.22 Circuit to illustrate timing.

TABLE 2.4 Delay Calculations

	A	*B*	*C*	*D*	*E*	*F*	*G*	Comments
t_0	X	X	X	0	X 0	X 1	X 0	Simulate *F*, schedule it for t_5
t_2	X	1	1	1	X 0	X 2	X 0	Simulate *E* and *F*, schedule *F* for t_7
t_4	0	1	0	1	X 1	X 3	X 0	Simulate *E* and *F*, schedule *E* for t_8, *F* for t_9
t_5	0	1	0	1	X 1	0 2	X 0	$F \leftarrow 0$, simulate *G*, no change
t_7	0	1	0	1	X 1	1 1	X 1	$F \leftarrow 1$, simulate *G*, schedule *G* for t_{11}
t_8	0	1	1	1	0 0	1 2	X 1	$E \leftarrow 0$, simulate *F* and *G* *G* unchanged, schedule *F* for t_{13}
t_9	0	1	1	1	0 0	0 1	X 2	$F \leftarrow 0$, simulate *G*, schedule *G* for t_{13}
t_{11}	0	1	1	1	0 0	0 1	1 1	$G \leftarrow 1$
t_{13}	0	1	1	1	0 0	1 0	0 1	$G \leftarrow 0$, $F \leftarrow 1$, schedule *G* for t_{17}
t_{17}	0	1	1	1	0 0	0 0	1 0	$G \leftarrow 1$

2.9.3 Evaluation Techniques

A number of techniques have been developed to evaluate response of the basic logic gates to input stimuli. For AND gates and OR gates, evaluation can be performed by looping on input values, two at a time, using AND and OR operations of the host computer's machine language instruction set. As we saw, it also works for ternary algebra. It is also possible to assign numerical values to ternary values as follows:

0-1
X-2
1-3

Then the AND of several inputs is the minimum value among all inputs and the OR is the maximum value among all inputs.

For binary values (i.e., no Xs), it is possible to count 1s on AND gates and count 0s on OR gates. If an *n* input AND gate has $n - i$ inputs at 1, for $i > 0$, then the output evaluates to 0. Whenever an input changes, the number of inputs having value 1 is incremented or decremented. If the number of inputs at 1 reaches *n*, the output is assigned the value 1. A similar approach works for an OR gate except it is necessary to count 0s.

Logic gates can also be evaluated using a truth table. This approach has the advantage that it will work with any circuit whose behavior can be described by a truth table. It is quite efficient when input values are grouped together in the descriptor cell so that the processing program can simply pick up the inputs field of the descriptor cell and use it to immediately index into a table that contains the output value corresponding to that input combination. It can also be used for ternary simulation or *n*-valued simulation. It requires $\log_2(n)$ bits for each input and the table can become excessively large but the simulation is quite rapid.

For logic gates such as the OR and the AND, three- or four-valued simulation requires two bits for each input. A six-input gate then requires a truth table, or lookup table, of 4096 two-bit entries. Only one table would be necessary because an AND (OR) gate with fewer than six entries could be computed by using the same table and filling on the left with 1s (0s). Furthermore, since AND and OR are both associative operations, a gate with more than six inputs could be computed using successive lookups.

The zoom table takes the truth table one step further. Rather than examine the function code to determine gate type, truth tables for the various functions are placed in contiguous memory. The function code is appended to the input values by placing the function code adjacent to the input values in the descriptor cell. Then, the catenated function/input value serves as an index into a much larger truth table to find the correct output value for a given function and set of inputs. The program implementation is more efficient because fewer decisions have to be made, one simple access to the value table produces the value regardless of the function.

For multiple-valued simulation, such as that described in Section 2.8 (Figure 2.18), two-dimensional lookup tables can be created based on the logic/strength levels used in the system. For example, if a 21-value system is used, then 21×21 lookup tables are created. The input values are used to create an index into that table. The index is used to retrieve the output response corresponding to these input values. For an n-input AND or OR gate, this process is repeated by means of a loop until all inputs have been evaluated.

Example For an AND gate with the number of inputs equal to "pincount" and with the value at input i stored at pinval(i), using a 21×21 lookup table, the C code used to evaluate the output response might appear as follows:

```
result = 1; // initialize result to 1
for (i = 0; i < pincount; i++) // loop through all inputs
  result = lookup_table + result * 21 + pinval(i);   ■■
```

2.9.4 Race Detection in Nominal-Delay Simulation

The zero-delay simulator resorted to multiple-value simulation to detect transient pulses caused by hazards. These unwanted signals are caused by delay in physical elements and can be detected by the nominal delay simulator using just the logic values {0,1} and individual element delay values—if the transients occur for nominal delay values. However, a hazard is only the possibility of a spurious signal, and the transient may not occur at nominal delay values. But, individual physical elements usually vary from nominal ratings; and some combination of real devices, each varying from its nominal value, may combine to cause a transient that would not have occurred if all elements possessed their nominal values. To further complicate matters, a transient may be innocuous or it may cause erroneous state transitions. In a circuit with many thousands of elements, how do we decide what delay values to simulate? Do we simulate only nominal delays? Do we also simulate worst-case delays?

Consider again the cross-coupled NAND latch. Erroneous behavior can occur if unintended pulses arrive at either the $\overline{Set}$ or $\overline{Reset}$ input. If the latch is cleared and a negative pulse of sufficient duration occurs on its $\overline{Set}$ line, it becomes set. Quite possibly, this situation will only occur for delay values that are significantly beyond nominal value. Furthermore, in a circuit with many thousands of gates there may only be a few asynchronous latches that are susceptible to glitches.

Potential problems can be addressed by identifying asynchronous latches, using the gate ordering technique described earlier. Then, with the latch inputs identified and grouped together, proceed with simulation. If a net changes value, and if that net is flagged as an input to an asynchronous latch, check other nets in that set for their most recent change. If another net previously changed within some user specified time range, a critical race may exist. The race exists if some combination of delay variances can combine to cause the first input change to occur later than the second input change. Therefore, trace the changing signals back to primary inputs or to a common origin. Increase the delay on all elements along the path to the latch input whose event occurred first. Decrease the delay on the elements along the path to the latch input that changed last, then resimulate. If this causes a reversal in the order in which the two inputs change, then a critical race exists.

Subsequent action depends on the reason for the simulation. For design verification, an appropriate course of action is to provide a message to the user advising either that primary input events are occurring too close together or that an event at a gate with fanout has caused a critical race. If patterns are being developed for the tester, then a state transition that is dependent upon the order in which two or more inputs change indicates a problem because it may be impossible to obtain repeatable tests on the tester. Many PCBs may respond correctly when tested, but every so often one or more fails. Attempts to isolate the problem can be frustrating because the individual components respond correctly when tested.

One possible solution is to alter the input stimuli by postponing one or more of the input stimuli changes to a later time period. This is sometimes referred to as *deracing*. If the race results from an event at a common fanout point, then somewhere along one of the two paths it may be possible to identify a gate by means of which an event can be inhibited. This is illustrated in Figure 2.23. An event reaches both the $\overline{Set}$ and $\overline{Clear}$ inputs of a latch. One path goes through an OR gate, the other path goes through other combinational logic. The event through the OR gate may be inhibited by first setting a 1 on the other input.

2.9.5 Min–Max Timing

The earliest and latest possible times at which a signal can appear at some point in a circuit can be determined through the use of min–max timing simulation. In this method each element is assigned a minimum and a maximum switching time. During simulation, these minimum and maximum times are added to cumulative earliest and latest times as the signal propagates through the circuit. The time interval between the earliest and latest times at which a signal switches is called the *ambiguity region*.

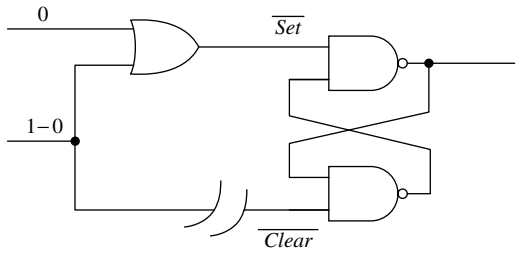


Figure 2.23 Blocking a propagation path.

The circuit in Figure 2.24 illustrates the computation of minimum and maximum delay values. The first block contains the numbers 0 and 10. These could represent the range of uncertainty as to when a signal arrives at a PCB from a backplane or from a tester due to skew caused by wiring, fixtures, and so on. The next block represents logic with a timing range of 20–30 ns, after which the circuit fans out to two other blocks. The upper path has a cumulative delay ranging from 25 to 47 ns by the time it arrives at the last block, and the bottom path has a cumulative delay of 40–70 ns. If the rightmost block represents an AND gate and if the signal arriving at the upper input is a falling signal, and the signal arriving at the lower input is a rising signal, then the numbers indicate a time region from 40 to 47 when there is uncertainty because the numbers imply that the lower input may rise as early as time 40 and the upper input may not fall until time 47.

A more careful analysis of the circuit reveals that there is a component 20/40 that is common to both signal paths. This component represents *common ambiguity.* If the common ambiguity is subtracted, it can be seen that the upper path will arrive at the AND gate no later than 7 ns after it fans out from the common element. The signal on the lower path will not arrive until at least 13 ns after the upper input change arrives. If this common ambiguity is ignored, then a pulse is created on the output of the gate and propagated forward when it could not possibly occur in the actual circuit. This pulse could result in considerable unnecessary activity in the logic forward of that point where the pulse occurred.

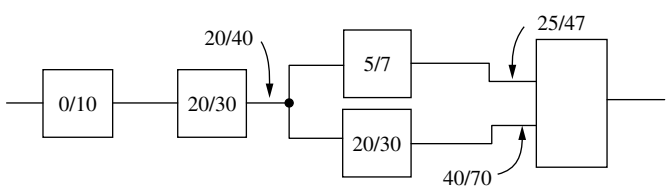


Figure 2.24 Min–max timing.

If the block on the right were an edge-triggered Delay flip-flop in which the upper input were the Data input and the bottom input were the Clock input, then results of the common ambiguity may be more catastrophic. With the common ambiguity, it is impossible to determine if the data arrived prior to the clock or after the clock. Hence, it would be necessary to set the flip-flop to X. To get accurate results, the common ambiguity must be removed.

A common ambiguity region can be identified with the help of the *causative link*.[21] This is simply a pointer included in the descriptor cell that points back to the descriptor cell of the element that caused the change. If two inputs change on a primitive and there is overlap in their ambiguity regions, then the simulator traces back through the causative links to determine if there is a common fanout point that caused both events. If a common source is found, then the ambiguity at the point is subtracted from the minimum and maximum change times of the two signals in question. If there is still overlap, then the block currently under consideration is set to X during the interval when the signals overlap if it is a logic gate or its state is set to X if it is a flip-flop.

2.10 SWITCH-LEVEL SIMULATION

Logic designers frequently find it necessary to simulate at different levels of abstraction. For a circuit containing hundreds of thousands, or millions, of gate equivalents, simulation at the RTL level is necessary. Simulation at a lower level of abstraction would require unacceptably long simulation times. However, on other occasions a more detailed simulation level may be required. For example, if a new function is created for a cell library, it may be designed at the transistor level and simulated at that level to ensure that it responds correctly. When satisfied that it is correct, it is added to the cell library and a gate or RTL level model may then be created for simulation purposes.

Consider the circuit in Figure 2.25, the intended function is $F = E \cdot (A + C) \cdot (B + D)$. But it was not designed by connecting AND and OR macrocells together! Rather, it was created by means of a transistor network in such a way that, depending on the values of A, B, C, D and E, there is always a connection from F to either V_{DD} or Gnd (but not both). So, is it correct? It is important to verify that the transistors have been connected correctly. The consequence of inserting such a design into a cell library with subtle errors could be catastrophic, possibly affecting more than one product release before being discovered.

The circuit in Figure 2.25 could be verified using Spice, an analog simulator that models circuits at the electrical level and uses continuous values to the accuracy possible (32 or 64 bits) on the host computer. For the small circuit in Figure 2.25, Spice would be acceptable. However, for much larger circuits, Spice simulations could require a great deal of CPU time. For such circuits, switch-level simulation often represents a reasonable compromise between analog and gate-level simulation, particularly when debugging.

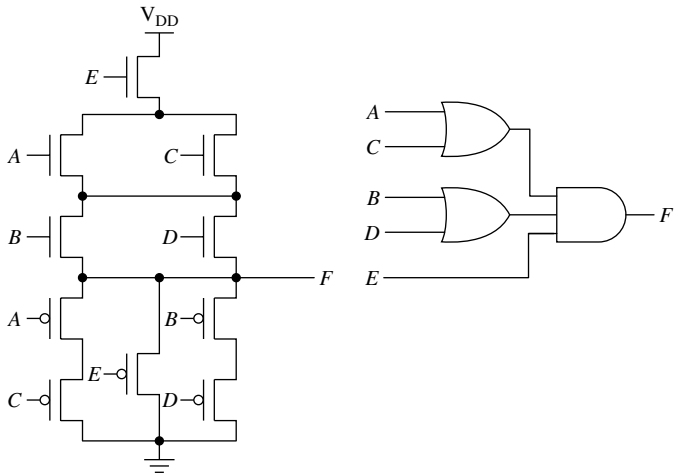


Figure 2.25 CMOS circuit.

Circuit behavior can, in general, be evaluated more rapidly when simulating at the logic level. For example, consider the circuit in Figure 2.25. If inputs *A,B,C,D,E* change from (0,1,1,0,1) to (1,1,1,0,1), an evaluation of one OR gate reveals that no event occurs beyond the inputs to that OR gate. While this provides faster simulation, when considering fault simulation, as will be seen in subsequent chapters, the switch-level model more accurately predicts circuit behavior in the presence of defects. Switch-level models can be accurately extracted from layout information, ignoring unimportant details while retaining circuit information that represents logic behavior. Hence modeling and simulation can be more precise at switch level than at gate level while running faster than a detailed electrical simulation using Spice.

Switch-level circuits are modeled as nodes connected by transistors that act as voltage-controlled switches. When turned on, a transistor connects two nodes; and when turned off, it isolates the nodes (i.e., the transistor acts as a very high resistance). If a node has sufficient capacitance, it can act as a memory device when isolated from all other nodes. This is known as dynamic memory. Other characteristics of switch-level circuits include bidirectional signal flow, resistance ratios, and charge sharing. The switch-level model uses discrete values to represent circuit elements and voltage levels, in contrast to Spice, which uses continuous values. This is accomplished by limiting the resistance and capacitance of the transistors to a small number of discrete values. The number of discrete values is just enough to permit representation of different circuit configurations, including transistor ratios, and resolution of their logic values in the presence of different signal values.

A switch-level model is a set of nodes $\{n_1, n_2, ..., n_n\}$ connected by a set of transistors $\{t_1, t_2, ..., t_m\}$. Each node n_i may be an *input node* or a *storage node*. Input

nodes are those such as V_{DD}, Gnd, data, and clock inputs that drive transistor source and drains. Storage node n_i has state $y_i \in \{0, 1, X\}$ and a size $\in \{k_1, ..., k_{max}\}$. The value X represents an uninitialized node or one whose logic value lies between 0 and 1, such as when values 1 and 0 are applied simultaneously to a node. Node sizes are ordered such that $k_1 < k_2 < \cdots < k_{max}$, where the ordering (<) denotes the capacitance of a node relative to other nodes. Input nodes have size ω, ($k_{max} < \omega$). The number of sizes, *max*, is arbitrary but chosen so as to permit all relative sizes to be correctly expressed. A node state is defined by the pair $<v,s>$, where v is the logic value and s is the signal strength. The transistor t_i has state $z_i \in \{0, 1, X\}$ and strength $\gamma \in \{\gamma_1, \gamma_2, ..., \gamma_{max}\}$, where the ordering $\gamma_1 < \gamma_2 < \cdots < \gamma_{max}$ indicates relative conductance. The state of a switch-level circuit is given by vectors $y = (y_0, y_1, ..., y_n)$ and $z = (z_0, z_1, ..., z_m)$. The excitation function E gives the steady-state response of the nodes for an initial set of node states when the transistors are held fixed in states determined by the initial node states

$$E(y) = F[y, z(y)] \tag{2.1}$$

where $z(y)$ denotes the vector of transistor states created when the nodes are in states given by the vector y. The operation of a switch-level circuit can be simulated by repeatedly computing the excitation states for the nodes and setting the nodes to these states until a stable state is reached. This is expressed as

$$y' = \lim_{k \to maxstep} E^k(y) \tag{2.2}$$

where *maxstep* denotes the maximum number of iterations. If the circuit has not stabilized at the end of *maxstep* steps, it may indicate oscillations in the circuit, which suggests that some of the nodes should be set to X.

When a signal passes through a transistor, its strength is determined by the transistor size. This is indicated in Figure 2.26(a), where State(V_{DD}) = <1, ω>, state(Gnd) = <0, ω> and state(A) = <0, ω> or <1, ω> depending on whether input A has logic value 0 or 1. Transistor T_1, a depletion mode transistor, is a pullup with strength γ_1, transistor T_2 has strength γ_2. The state at node Z is determined by the *connection function*.[22] The first step in determining the state at a node is to find the strength of the strongest applied signal(s). When $A = 1$, applied signals from V_{DD} and Gnd converge at Z with strength γ_1 and γ_2. Since there are two signals driving node Z, the signal value v at Z must be resolved. The set W of all applied signals with maximum applied strength is formed. In this example, state γ_2 is the strongest signal and it has a single source. Once the set W is formed, the following rules apply:

If W contains X or it contains both 0 and 1, then v = X.

If W contains 0 but does not contain 1 or X, then $v = 0$.

If W contains 1 but does not contain 0 or X, then $v = 1$.

If W does not contain 0, 1, or X, then v = Z.

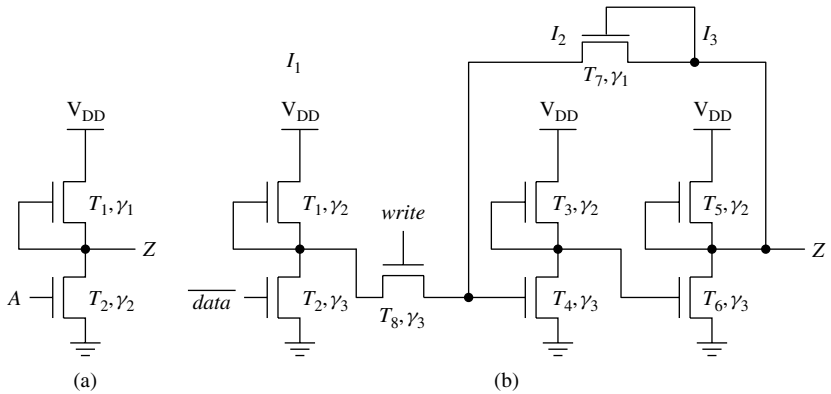


Figure 2.26 Switch-level circuits.

In this example, $W = \{0\}$, so $v = 0$. This operation is often denoted by the # operator, and it uses the lattice structure depicted in Figure 2.27. In this structure, $<0, \kappa_i>$ and $<1, \kappa_i>$ resolve to $<X, \kappa_i>$, whereas $<0, \kappa_i>$ and $<e, \kappa_j>$, $e \in \{0,1,X\}$, $j < i$, resolves to $<0, \kappa_i>$. In general the higher strength, often called the least upper bound (lub), prevails.

Figure 2.26(b) contains a somewhat more complex circuit: It is a static RAM cell made up of two cross-coupled inverters, I_2 (T_3 and T_4) and I_3 (T_5 and T_6). The $\overline{data}$ signal is inverted by inverter I_1 (T_1 and T_2). If *write* is high, then the signal *data* passes through T_8 where it shares a common node with transistor T_7. But T_7 has strength γ_1 and the *data* signal appearing at T_8 may have strength γ_2 or γ_3. In either case its strength is stronger than that of the signal coming from T_7, so *data* will control I_2, which in turn controls I_3. Note that if *write* = 0, then the value coming from T_8 is Z, so the signal coming from T_7 will control T_4.

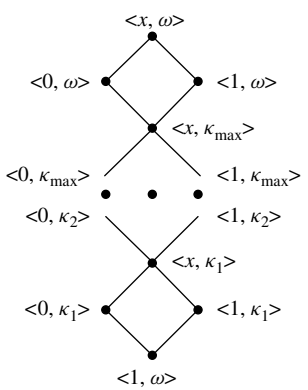


Figure 2.27 Lattice representation of the # operator.

So far, calculations have been intuitive. However, to implement a simulator capable of evaluating circuit behavior in response to applied stimuli, it is necessary to define processing rules that anticipate all circumstances. For logic simulation, where the elements are unidirectional, evaluation can consist of repeated table lookups until the output response is resolved. In fact, if the circuit is expressed in terms of unidirectional transistors (e.g., the Verilog nmos, pmos, and cmos primitives), simple extensions to the gate-level simulator are sufficient.

However, when a circuit is modeled in terms of the Verilog tran, tranif0 and tranif1, rtran, rtranif0, rtranif1 primitives, a gate-level simulator is no longer adequate. As can be seen from Figure 2.26(b), some nodes are driven by two or more transistors. The problem is compounded by the fact that the transistors have different strengths. The state at a node can be calculated using the connection function, but with a large number of bidirectional transistors, an event at a node could propagate through many transistors, each event necessitating numerous additional calculations.

Early attempts at solving the problem of simulating switch-level elements attempted to extend the capabilities of the gate-level simulator. One artifice to achieve this modeled the bidirectional transistor as a pair of unidirectional transistors connected back-to-back.[23] Unfortunately, the two transistors can form a cycle in which signals become trapped. This is seen in Figure 2.28. In Figure 2.28(a) the transistor controlled by input *A* is bidirectional, whereas in Figure 2.28(b) it has been replaced by two unidirectional transistors with signal direction denoted by the arrows.[24] In Figure 2.28(a) the value at *D* is <4, 0>. Let *B* switch from 1 to 0. The path from Gnd to *C* is blocked, so the contribution from the lower transistor, controlled by input *B*, is Z. However, from Figure 2.28(b) it can be seen that one of the back-to-back transistors controlled by input *A* is driving node *C* with state <4, 0> and the other transistor is driving node *D* with <4, 0>. As a result the depletion transistor, with a strength of 3, cannot alter the value at *D* and so the output of the NAND circuit is 0 when it should be 1.

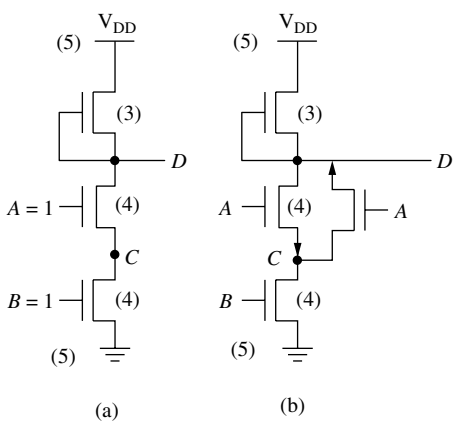


Figure 2.28 Trapped signal.

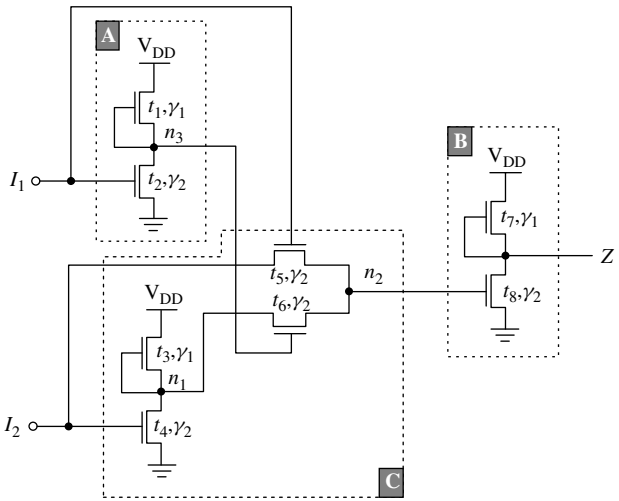


Figure 2.29 Partitioned network.

A large transistor network described in terms of bidirectional transistors, such as the Verilog tran, tranif0 and tranif1, can be quite confusing to analyze, even for humans who can employ experience, intuition, and pattern recognition to decompose a network into smaller subcircuits with recognizable features. For the computer this human process must be replaced by a series of precise, methodical steps before the computer can analyze and determine the behavior of the circuit. The first step in this process is partitioning.

Two partitioning schemes have been devised, they are referred to as *static partitioning* and *dynamic partitioning*. Static partitioning breaks a circuit into components by cutting the leads that drive the gates. This is illustrated in Figure 2.29, where a transistor network has been broken into three components, referred to as *channel connected components*, labeled A, B, and C. The connection from transistors t_1 and t_2 to transistor t_6 is cut, so t_1 and t_2 become a standalone component labeled A. Also, the connection from t_5 and t_6 to transistor t_8 is cut, causing t_7 and t_8 to become a separate component labeled B. The remaining four transistors, t_3, t_4, t_5, and t_6, become component C. The second way to partition, dynamic partitioning, uses the logic values on the transistor gates. If the value on a gate is 0, then the transistor, for evaluation purposes, is nonexistent. However, this method requires that the circuit be repeatedly partitioned as node values change in response to events on input nodes.

Note that because individual components are evaluated independently from the rest of the circuit, it is quite straightforward to merge switch-level simulation with gate- and RTL-level simulation. Evaluating individual components can become complicated, but the components themselves become unidirectional elements, so in their interactions with other circuit components they can be scheduled like logic

gates. If an event occurs on one or more inputs, the component is evaluated, and if one or more of its outputs change, the components driven by the changing output(s) are evaluated.

Component evaluation is based on events appearing at both of the original circuit inputs, these would be I_1 and I_2 in Figure 2.29, and the inputs created by partitioning. Component A has a single input, I_1. Component B also has a single input, the wire driving the gate of transistor t_8. That wire is also an output of component C. The inputs to component C are I_2 and the two wires driving transistors t_5 and t_6.

In order to evaluate a component and find its steady state, it is necessary to find, for a set of signal values applied to the input pins of the circuit, a set of steady-state values v_i at internal nodes n_i such that $v = f(v)$. From Eq. (2.2) it was seen that this could require as many as maxstep iterations. The solution v is referred to as the least fixed point of f. The discussion here, from Bryant,[25,26] characterizes the problem by means of the following expression:

$$v = \mathrm{E} * x \vee y \vee \mathrm{G} * v \tag{2.3}$$

where v is the minimum set of steady-state signals satisfying the equation. In Eq. (2.3) E is a matrix in which e_{ij} equals the strength of the strongest transistor connecting storage node n_i and input node i_j or 0 if no such transistor exists. The component x_j of vector x is equal to ω if input node i_j is 1, or λ if input node i_j is 0. The components y_j of vector y represent the size of node n_j. The matrix G describes the interconnections of the storage nodes; that is, g_{ij} is equal to the strength of the strongest transistor connecting nodes n_i and n_j. The operator $\vee$ is the least upper bound (lub) operation and $*$ denotes matrix multiplication. In matrix multiplication, individual elements are multiplied using the operator $\cap$, where $a \cap b$ denotes the minimum of a and b, and addition of the resulting product terms is accomplished using the lub $\vee$.

Equation (2.1) is solved iteratively until it stabilizes—that is, until $v = f(v)$. Note that in this equation the value at node n_i represents the combined effect of

1. The direct connection to each input node i_j as determined by $e_{ij} \cap x_j$
2. The initial charge y_i at node n_i
3. The connections $g_{ij} \cap v_j$ from node n_i to other nodes in the circuit

What happens when the circuit contains Xs? Before addressing this question, some definitions are in order. The vectors $\boldsymbol{a}$ and $\boldsymbol{b}$ obey the ordering $\boldsymbol{a} \leq \boldsymbol{b}$ iff $a_i \leq b_i$ (that is, $a_i < b_i$ or $a_i = b_i$) for all i. The lub of a set of signals $\in \{0, 1, \mathrm{X}\}$ equals 1 (0) iff all elements of the set are 1 (0), else it is X. Consider the mapping $f\colon\ B^n \rightarrow T^m$, its ternary extension is defined as the function $f^t\colon\ T^n \rightarrow T^m$ such that

$$f^t(\boldsymbol{a}) = \mathrm{lub}\{f(\boldsymbol{b}) | \boldsymbol{b} \in B^n, \boldsymbol{b} \leq \boldsymbol{a}\}$$

Expressed in words, when some inputs to f^t equal X, then each output assumes a Boolean value iff it would assume this value for all possible combinations of 0s and 1s. In the following matrix equations, that is essentially what the equations for u and d provide.

$$\mathrm{r} = \mathrm{E}^{\min} \cdot \|x\| \uparrow \|y\| \uparrow \mathrm{G}^{\min} \cdot \mathrm{r} \tag{2.4}$$

$$\mathrm{u} = \mathrm{block}(\mathrm{E}^{\max} \cdot \lceil x \rceil \uparrow \lceil y \rceil \uparrow \mathrm{G}^{\max} \cdot \mathrm{u}, \mathrm{r}) \tag{2.5}$$

$$\mathrm{d} = \mathrm{block}(\mathrm{E}^{\max} \cdot \uparrow \lfloor y \rfloor \uparrow \mathrm{G}^{\max} \cdot \mathrm{u}, \mathrm{r}) \tag{2.6}$$

In these equations, $\|a\|$ denotes the strength of a, $\lceil a \rceil$ denotes the strength of a if a has state 1 or X, and 0 otherwise, and $\lfloor a \rfloor$ denotes the strength of a if a has state 0 or X, and 0 otherwise. The operator $\uparrow$ yields the maximum of its arguments, and the dot $(\cdot)$ denotes matrix multiplication with $\cap$ corresponding to element multiplication and $\uparrow$ corresponding to addition. Given two strength values a and b, block(a,b) equals a if $a \geq b$ and it equals 0 otherwise. The matrices $\mathrm{E}^{\min}$ and $\mathrm{G}^{\min}$ represent the matrices E and G, but with the proviso that transistors in the X state have 0 conductance. Conversely, $\mathrm{E}^{\max}$ and $\mathrm{G}^{\max}$ represent the matrices E and G but with transistors in the X state assumed to be fully conducting.

A node n_i will have state 1 iff no combination of transistor conductances could cause the node to assume the value 0 or X. This implies that $d_i = 0$. Likewise, n_i will have target state 0 iff $u_i = 0$. As a result, the value at node n_i is determined to be

$$n_i = \begin{cases} 1 & \text{if } \mathrm{d}_i = 0 \\ 0 & \text{if } \mathrm{u}_i = 0 \\ \mathrm{X} & \text{otherwise} \end{cases} \tag{2.7}$$

Example Component C of Figure 2.29 will be used to illustrate the evaluation process. Initial input values will be I_1, $I_2 = (0, 1)$. The first step will be to evaluate Eq. (2.4) for r.

$$\mathrm{r} = \begin{bmatrix} \lambda & \gamma_2 & \lambda \\ \lambda & \lambda & \lambda \end{bmatrix} \cdot \begin{bmatrix} \omega \\ \omega \\ \omega \end{bmatrix} \uparrow \begin{bmatrix} \kappa_1 \\ \kappa_1 \end{bmatrix} \uparrow \begin{bmatrix} \lambda & \gamma_2 \\ \gamma_2 & \lambda \end{bmatrix} \cdot \begin{bmatrix} \mathrm{r}_1 \\ \mathrm{r}_2 \end{bmatrix} \tag{2.8}$$

Component C has three input nodes, $\mathrm{V_{DD}}$, Gnd, and I_2, and two storage nodes, n_1 and n_2. The matrix E indicates a connection between Gnd and n_1, as a result of I_2 having value 1. There are no other direct connections between the input nodes and the storage nodes. All three of the input nodes have strength ω. The strengths of the storage nodes are set to κ_1. The matrix G reflects the fact that transistor t_6 is conducting, because node n_3 is a 1 (it is the complement of I_1). Therefore a connection

exists between n_1 and n_2. Note also that the matrix G is symmetric. Equation (2.8) reduces to

$$\begin{bmatrix} r_1 \\ r_2 \end{bmatrix} = \begin{bmatrix} \gamma_2 \\ \lambda \end{bmatrix} \uparrow \begin{bmatrix} \kappa_1 \\ \kappa_1 \end{bmatrix} \uparrow \begin{bmatrix} \gamma_2 \downarrow r_2 \\ \gamma_2 \downarrow r_1 \end{bmatrix} \tag{2.9}$$

At this point it is necessary to make use of the following equation:

$$r = \lim_{k \to \infty} f_s^{\,k}(0) \tag{2.10}$$

Equation (2.10) asserts that r can be solved by initializing r_1 and r_2 to 0 and then solving iteratively until a steady state is reached. That yields

$$\begin{aligned} r_1 &= 0 \quad \gamma_2 \quad \gamma_2 \quad \gamma_2 \\ r_2 &= 0 \quad \kappa_1 \quad \gamma_2 \quad \gamma_2 \end{aligned}$$

It still remains to solve Eqs. (2.5) and (2.6) for u and d. Note that E, E^{min}, and E^{max} are identical because none of the inputs or storage nodes are at X. The same is true for G, G^{min}, and G^{max}. The matrix $\lceil x \rceil$ evaluates to $[\omega\, 0\, \omega]^T$ so u becomes

$$\begin{bmatrix} u_1 \\ u_2 \end{bmatrix} = \text{block}\left(\begin{bmatrix} \lambda \\ \lambda \end{bmatrix} \uparrow \begin{bmatrix} \kappa_1 \\ \kappa_1 \end{bmatrix} \uparrow \begin{bmatrix} \gamma_2 \downarrow u_2 \\ \gamma_2 \downarrow u_1 \end{bmatrix}, \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} \right) \tag{2.11}$$

For convenience, let u = block(v, u) and d = block(e, d). Setting $v_1 = v_2 = 0$ and then iterating, we obtain

$$\begin{aligned} v_1 &= 0 \quad \kappa_1 \quad \kappa_1 \\ v_2 &= 0 \quad \kappa_1 \quad \kappa_1 \end{aligned}$$

Solving for e is similar, except that $\lfloor x \rfloor$ becomes $[0\ \omega\ 0]^T$. Thus,

$$\begin{aligned} e_1 &= 0 \quad \gamma_2 \quad \gamma_2 \quad \gamma_2 \\ e_2 &= 0 \quad \lambda \quad \gamma_2 \quad \gamma_2 \end{aligned}$$

This results in

$$\begin{aligned} u_1 &= \text{block}\,(v_1, r_1) = \text{block}(\kappa_1, \gamma_2) = 0 \\ u_2 &= \text{block}\,(v_2, r_2) = \text{block}(\kappa_1, \gamma_2) = 0 \end{aligned}$$

$$d_1 = \text{block}\ (e_1, r_1) = \text{block}(\gamma_2, \gamma_2) = \gamma_2$$

$$d_2 = \text{block}\ (e_2, r_2) = \text{block}(\gamma_2, \gamma_2) = \gamma_2$$

From Eq. (2.7) it follows that $n_1 = n_2 = 0$, so the output of component C is 0. That becomes an input to component B, where it gets inverted, so $Z = 1$. ■■

This small example required a large number of mathematical computations in order to achieve a final steady state. While it provides a theoretical basis for switch-level simulation, it is not practical. In practice, simulation programs that compute next state for a switch-level circuit bear a resemblance to those used in gate-level simulation. This will be illustrated using the switch-level algorithm adapted from Bose et al.[27]

We start with some definitions. A transistor is in the *indefinite state* if the value on its gate is X. A *path* in a channel-connected component is a set of transistors in which the source (drain) of one is connected to the drain (source) of another transistor in the set. A *definite path* is one in which no transistors are in the indefinite state. The strength of a signal along a path is the minimum of the signal strength at the path source and the minimum strength transistor along the path. A path is *blocked* at node i if i is the destination of a stronger path. A *downgoing path* originates at a source node with logic value 0 or X, whereas an *upgoing path* originates at a source node with logic value 1 or X.

The strength of the strongest downgoing definite path to node i that is unblocked at all nodes prior to i is denoted $\text{def}_{0,i}$. The strongest downgoing path, definite or indefinite, to node i that is unblocked at all nodes prior to i is denoted $\text{indef}_{0,i}$. The strongest upgoing paths are denoted similarly, that is, $\text{def}_{1,i}$ and $\text{indef}_{1,i}$. The maximum strength of the signal flow through transistor j connecting nodes p and q is denoted $\text{sw\_max}_{v,j}$, where $v \in \{0,1\}$. Given a switch-level circuit with n nodes, the algorithm follows:

```
// initialize nodes
  for (all nodes i)
    if y_i ∈ {0,X} then def_{0,i} = κ_i
    else def_{0,i} = λ
  for (all nodes i)
    if y_i ∈ {1,X} then def_{1,i} = κ_i
    else def_{1,i} = λ
  for (all transistors connecting nodes n and m)
    sw_max_{0,t} = sw_max_{1,t} = λ
// compute strongest definite paths to nodes
  for (all strengths s in decreasing order)
    for (each i with def_{0,i} = s and s ≥ def_{1,i})
      for (each "on" transistor t connecting i to m)
        if def_{0,m} does not dominate min(s,σ_t)
```

```
          set sw_max0,t to max(sw_max0,t, min(s, σt))
      set def0,m to max(def0,m, min(s, σt))
  for (each i with def1,i = s and s ≥ def0,i)
    for (each "on" transistor t connecting i to m)
      if def1,m does not dominate min(s,σt)
        set sw_max1,t to max(sw_max1,t, min(s, σt))
      set def1,m to max(def1,m, min(s, σt))
// quit early if no transistor is indefinite
  if all transistors are definite
    for (all nodes i)
      if def0,i dominates def1,i then set yi to 0
      else if def1,i dominates def0,i then set yi to 1
      else set yi to X
```

Example Given the circuit in Figure 2.30, assume that V_{DD} and Gnd have strength 7, and the transistors have strengths between 3 and 6, as indicated. The storage nodes all have strength 1; with the exception of the output F (node n_3), it has strength 2. The values on the gate inputs are $A,B,C,D,E = (0,1,1,0,1)$. The pairs of numbers in the figure represent the values $\text{sw\_max}_{0,i}$ and $\text{sw\_max}_{1,i}$. So, for example, through the NMOS transistor E (connected to V_{DD}), the strength of the 1 signal is 6, while the strength of the 0 signal is 0. Since the NMOS transistor A is turned off, both the 0 and 1 signals through A are 0. Note, however, that the PMOS transistor A is on, so from node n_3 there is an upgoing signal of strength 3 through A. The PMOS transistor D is on, so the strength of the 0 signal is 5 and the strength of the 1 signal is 0. The remaining transistors are evaluated similarly.

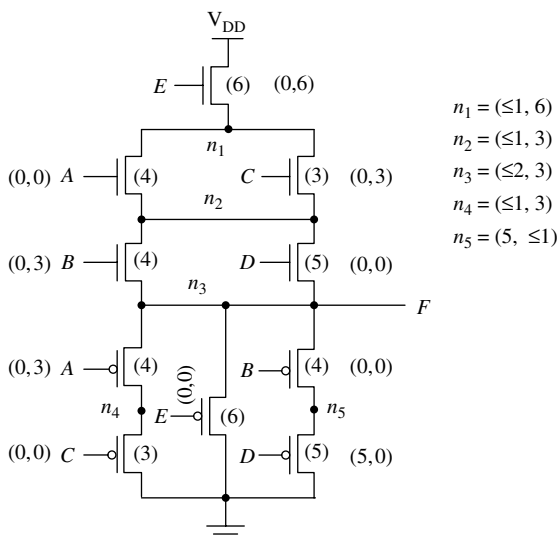


Figure 2.30 Computing node signals.

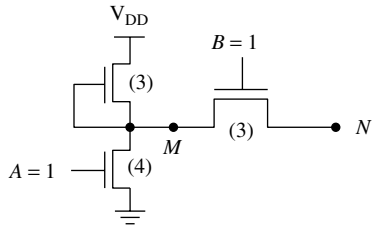


Figure 2.31 Problems from evaluation ordering.

The definite pairs $def_{0,i}$ and $def_{1,i}$ are listed to the right of the drawing. For node n_1 the values are (≤1, 6). Since the NMOS transistor *E* is turned on, the upgoing signal provided by V_{DD} is equal to the strength of transistor *E*, which is 6. There is no downgoing path to node n_1 from any transistor, so the 0 strength of node n_1 is, at most, the node strength, which is 1. The strongest upgoing signal to node n_2 comes from transistor *C*. It has strength 3. The remaining nodes are evaluated similarly. Because there is an upgoing path of strength 3 to the output node *F*, and a downgoing path of strength ≤2 to node F, the output resolves to a logic 1.

Note that the algorithm calls for processing nodes in decreasing order of strengths. The reason for this can be seen in this next example. ■■

Example Figure 2.31 contains an inverter with an output transistor *B*.[24] Start by propagating the signal from V_{DD}. It causes the signal <3, 1> to appear at the output.

Now consider what happens when the signal from Gnd is processed.The signal at Gnd appears at node *M* as <4, 0>. This signal is attenuated as it passes through B to become <3, 0> at output *N*. Now the two signals <3, 1> and <3, 0> are resolved to X at *N*.

When Gnd is processed first, the signal <4, 0> appears at *M*. It is then propagated through *B*, to the output, where it is attenuated to become <3, 0>. The signal from V_{DD} is processed next. It reaches M, where it appears as <3, 1>. The signals <4, 0> and <3, 1> at *M* resolve to <4, 0>. That signal is attenuated through transistor *B* to become <3, 0> at *N*. ■■

Up to this point, no mention has been made of what to do when Xs are encountered. In the discussion of matrix calculations, the matrices u and d identify nodes that conflict, and those that converge, when Xs are present. The conflicting nodes are set to X, and the nodes that converge are set to the converged value. In the algorithm described here, the extension of the algorithm for indefinite paths performs a similar function:

```
for (all nodes i)   // compute strengths of indefinite
                    // paths to nodes
  initialize indef0,i to def0,i
```

```
    initialize indef1,i to def1,i
  for (all strengths s in decreasing order)
    for (each i with indef0,i = s and s ≥ def1,i)
      for (each "on" or "indefinite" transistor t
         connecting i to m)
        if indef0,m does not dominate min(s,σt)
          set sw_max0,t to max(sw_max0,t, min(s, σt))
        set indef0,m to max(def0,m, min(s, σt))
    for (each i with indef1,i = s and s ≥ def0,i)
      for (each "on" or "indefinite" transistor t
         connecting i to m)
        if indef1,m does not dominate min(s,σt)
          set sw_max1,t to max(sw_max1,t, min(s, σt))
        set indef1,m to max(indef1,m, min(s, σt))
  for (all nodes i)     // compute new logic values of
                        // nodes
    if def0,i dominates indef1,i       then set yi to 0
    else if def1,i dominates indef0,i then set yi to 1
    else set yi to X
```

2.11 BINARY DECISION DIAGRAMS

Binary decision diagrams (BDDs) provide a means for representing circuit behavior by means of graphs. In recent years they have grown in importance because of their applicability to several areas of digital design, including simulation, automatic test pattern generation, synthesis, and design verification. Here we discuss their application to simulation—in particular, cycle simulation (see Section 2.12). In subsequent chapters we discuss their application to other areas of electronic design automation (EDA).

2.11.1 Introduction

Binary decision diagrams were introduced by Sheldon Akers in 1978.[28] Akers' work was based on research into binary decision programs by C. Y. Lee.[29] BDDs can be used to represent Boolean expressions in a form that resembles a decision tree. BDDs are implementation-free, they can determine the response of a circuit to input stimuli but offer no insight into the structure of the circuit. This can be considered an advantage, because it permits circuits described at very different levels of abstraction to be compared for equivalence.

We start with some basic definitions, derived from Aho et al.[30] A *graph G = (V, E)* is a finite, nonempty set of *vertices V* and a set of *edges E*. The edges are pairs of vertices (v_1, v_2) where $v_1, v_2 \in V$. If the edges are ordered pairs, then the graph is said to be a *directed graph*. In a directed graph the edge (v_1, v_2) is said to be from v_1 to v_2,

where v_1 is called the *tail* and v_2 is the *head*. A *path* is a sequence of edges of the form (v_1, v_2), (v_2, v_3), ..., (v_{n-1}, v_n). The path is from v_1 to v_n, and is of length $n - 1$. A cycle is a path that begins and ends at the same vertex.

A directed graph with no cycles is called a *directed acyclic graph (DAG)*. A *tree* is a DAG that satisfies the following properties:

1. There is exactly one vertex, called the *root*, which no edges enter.
2. Every vertex except the root has exactly one entering edge.
3. There is a unique path from the root to each vertex.

If $(v_1, v_2) \in V$, where V is a tree, then v_1 is the *parent* of v_2 and v_2 is the *child* of v_1. A vertex with no descendents is called a *terminal vertex*, also called a *leaf*; the remaining vertices are called *nonterminal vertices*. If a path exists from v_i to v_j, then v_i is an *ancestor* of v_j, and v_j is a *descendent* of v_i. An *ordered tree* is one in which some ordering rule is imposed on the children of each vertex. A *binary tree* is an ordered tree in which each vertex v has at most two children, denoted *low(v)* and *high(v)*. The edge from vertex v to low(v) corresponds to the value $v = 0$ and is sometimes called the *0-edge*. Likewise, the edge leading to high(v) corresponds to the value $v = 1$ and is sometimes called the *1-edge*. A nonterminal vertex v has associated with it an attribute *index(v)* $\in \{1, 2, ..., n\}$. A terminal vertex v has as attribute a value *value(v)* $\in \{0, 1\}$.

The number of vertices in a binary decision tree grows exponentially. A tree generated from three variables $\{x_1, x_2, x_3\}$ has seven nonterminal vertices and eight terminal vertices. In general, a binary decision tree has $2^n - 1$ nonterminal vertices and 2^n terminal vertices. This does not represent any appreciable savings over the corresponding truth table with its 2^n rows. However, a *binary decision diagram* (*BDD*) offers significant potential savings. It permits many edges to terminate at any given vertex. One immediately obvious gain is in the representation of the terminal vertices. When all the terminal vertices have value 0 or 1, then there only need be two terminal vertices, one with value 0 and the other with value 1. A computer program used to represent the function can immediately free up $2^n - 2$ structures used to represent the terminal vertices.

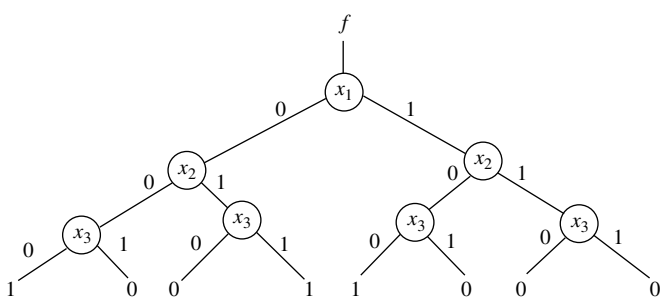


Figure 2.32 Binary decision tree.

Example Consider the binary tree in Figure 2.32. It corresponds to the equation

$$f = \bar{x}_1 \cdot \bar{x}_2 \cdot \bar{x}_3 + \bar{x}_1 \cdot x_2 \cdot x_3 + x_1 \cdot \bar{x}_2 \cdot \bar{x}_3$$

The complete truth table corresponding to this BDD is

x_1	x_2	x_3	f
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	0

■■

Note that in the binary decision tree the vertices are labeled x_i, with the root vertex labeled x_1. In the discussion that follows, we will often label a vertex solely with the subscript, which serves as its index. When using subscripts of the x_i as indices, indices of descendents will appear in ascending order; that is, if vertex v is nonterminal, we require index(v) < index(low(v)) and index(v) < index(high(v)).

To evaluate a function for particular values of x_1, x_2 and x_3 in a truth table, search down the truth table until matching values are found, then look for the value of the function in the rightmost column of the same row. To evaluate a function using a BDD, start at the root and follow the 0- and 1-edges corresponding to the binary values assigned to the variables. For example, if x_1 is 1, x_2 is 0 and x_3 is 1, then take the 1-edge from vertex x_1 to vertex x_2, take the 0-edge from vertex x_2 to vertex x_3, and take the 1-edge out of x_3. This process terminates at a vertex assigned the value 0.

This BDD was generated by arbitrarily assigning variable x_1 as the root and creating a 0-edge and a 1-edge from that root. This causes two *subgraphs* to be created. In each of these subgraphs the variable x_2 serves as the root. This process can be repeated at the subgraphs with root x_2. Further iterations eventually lead to terminal vertices, with terminal values matching the values in the truth table entry corresponding to the edge values on the path from the root to the given terminal vertex.

The reader may recognize this as a repeated application of Shannon's expansion:

$$f(x_1, x_2, ..., x_i, ..., x_n) = x_i \cdot f(x_1, x_2, ..., 1, ..., x_n) + \bar{x}_i \cdot f(x_1, x_2, ..., 0, ..., x_n)$$

For the equation given in the example above, the first application of Shannon's expansion yields the results shown in Figure 2.33. Note that it is not necessary to create the truth table for a Boolean expression. Continued applications of Shannon's expansion will yield the binary decision tree shown in Figure 2.32.

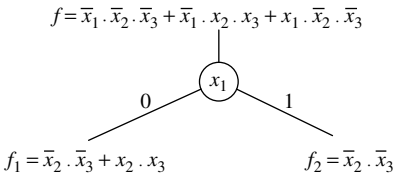

$f = \overline{x}_1 . \overline{x}_2 . \overline{x}_3 + \overline{x}_1 . x_2 . x_3 + x_1 . \overline{x}_2 . \overline{x}_3$
x_1
0
1
$f_1 = \overline{x}_2 . \overline{x}_3 + x_2 . x_3$
$f_2 = \overline{x}_2 . \overline{x}_3$

Figure 2.33 Applying Shannon's expansion.

The BDD in Figure 2.32 was drawn in such a way that there was a terminal corresponding to every entry in the truth table. However, many of the branches are unnecessary. For example, the rightmost path $(x_1, x_2) = (1,1)$ leads to x_3, but both terminal vertices emanating from x_3 are 0, regardless of whether x_3 is 0 or 1. This branch of the tree can be pruned and the 1-edge from x_2 can terminate with a 0. Another way to shorten the graph is to represent the terminal vertex as x_3 or $\overline{x}_3$. This produces the BDD shown in Figure 2.34(a). Note that a BDD can be redrawn with any variable as the root. This often yields significantly different BDDs, as seen when comparing Figures 2.34(a) and 2.34(b), which represent the same function.

This process can be reversed. A sum-of-products Boolean equation can be derived from the BDD. First, label the branches emanating from x_1 as f_1 and f_2. Then, f can be expressed as $f = \overline{x}_1 \cdot f_1 + x_1 \cdot f_2$. Pursuing this a step further, vertex f_1 can be represented as $f_1 = \overline{x}_2 \cdot g_1 + x_2 \cdot g_2$ and vertex f_2 can be represented as $f_2 = \overline{x}_2 \cdot h_1 + x_2 \cdot h_2$. From Figure 2.34 it can be seen that $g_1 = \overline{x}_3$, $g_2 = x_3$, $h_1 = \overline{x}_3$, and $h_2 = 0$. From here, the minterms for f are readily obtained (a *minterm* is a sum-of-products term in which every variable appears in true or complement form). The maxterms can be found by tracing all paths to leafs with value 0 (a *maxterm* is a product-of-sums term in which every variable appears in true or complement form).

Some useful BDDs are illustrated in Figure 2.35. The D flip-flop in Figure 2.35(a) retains its existing value if the clock, C, is 0. If C is 1 (and, assuming a positive edge), then the value at the D input is transferred to the output Q. The formula for this operation is $Q^{k+1} = Q^k \overline{C}^k + D^k C^k$. Behavior of the toggle flip-flop in Figure 2.35(b) obeys the formula $Q^{k+1} = C^k T^k \overline{Q}^k + \overline{C}^k Q^k + \overline{T}^k Q^k$.

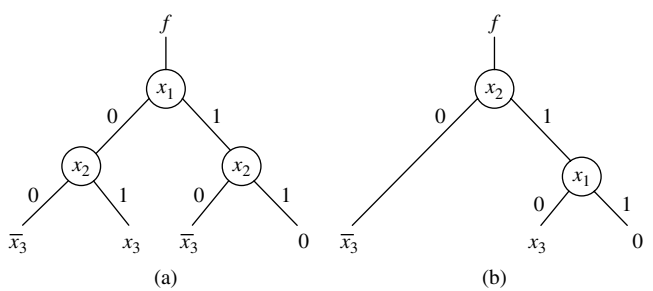


Figure 2.34 Reduced BDD.

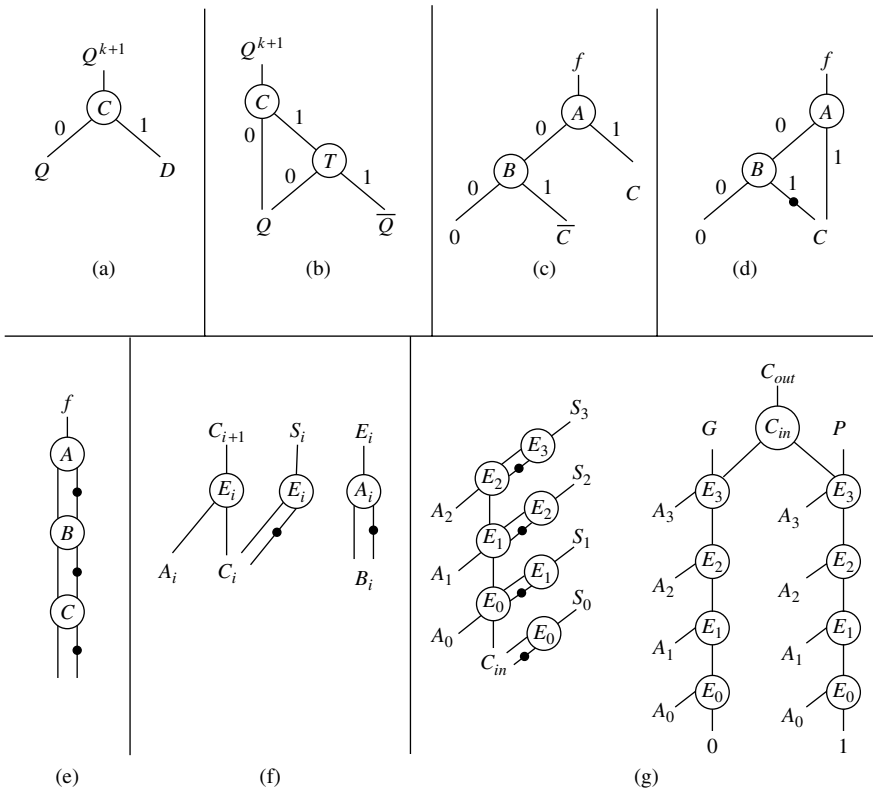


Figure 2.35 Some useful BDDs.

Figure 2.35(c) implements the equation $f = \overline{A} \cdot B \cdot \overline{C} + A \cdot C$. Figure 2.35(d) implements the same equation, but two new concepts are introducted in this BDD. First, the right branch exiting from A now goes straight down and shares the variable C with variable B. Second, there is a bubble on the edge emanating from B and terminating on C. This bubble is used to indicate that the value is to be complemented. So, if the BDD is traversed from the entry point at the top, through the left branch emanating from A, and then through the right branch emanating from B, the final result f is not C, but rather $\overline{C}$; for example, if C is 0, then $f = 1$. The general rule is: If there are an odd number of bubbles (inversions) in the path from the entry point to the terminal vertex, the result is complemented. If an even number of bubbles are encountered, the result is not inverted.

Figure 2.35(e) illustrates the BDD for the expression $f = A \oplus B \oplus C$. In this example, both edges emanating from A terminate at vertex B, and the edges emanating from B both terminate on vertex C. It clearly illustrates the rule concerning the number of inversions mentioned in the previous paragraph. The BDDs in Figure 2.35(f) represent a full-adder; they illustrate yet one more new concept. The

edge out of C_{i+1} terminates on E_i. But E_i is not an input variable. It is an intermediate variable whose value is calculated using the rightmost BDD in Figure 2.35(f). Thus, if $A_i = B_i = 1$ and $C_i = 0$, then E_i is determined to be 0. So, when calculating the sum S_i and carry C_{i+1}, the left branch is taken out of E_i in both BDDs to get a carry of 1 and a sum of 0.

In Figure 2.35(g) all of these concepts are combined to get a complete set of BDDs for a 4-bit adder with carry look-ahead (CLA). The values for the E_i are obtained from the BDD in Figure 2.35(f). To connect several of these together to represent a 16- or 32-bit data path, it would be necessary to develop a BDD for a CLA. The inputs to the CLA will be driven by the propagate (P), generate (G), and C_{out} outputs from the BDD in Figure 2.35(g).

Given a set of values assigned to the inputs of a circuit, BDDs can be used to compute the circuit response to that set of values. The BDD can be stored in a data structure using pointers. From the root this BDD can be traversed in a programming language like C or C++ quite easily to obtain the circuit response to a given set of inputs. Consider, for example, the reduced BDD in Figure 2.34(b). If $x_2 = 0$, the value of the expression is immediately determined to be equal to $\bar{x}_3$. Compare that with the number of programming steps required to evaluate an RTL expression representing the three original minterms. First, the variables have to be complemented. Then, two AND operations are required to evaluate each minterm. Finally, the results for the three minterms have to be ORed together to produce the final result. For event-driven simulation the comparison becomes more complex because the number of computations depends on how many inputs change and how far the events propagate through the circuit. There is a fixed overhead associated with creating the initial BDDs in storage, but for large circuits with many input vectors, that represents a small percentage of the overall computation time.

2.11.2 The Reduce Operation

In the discussion that follows we examine some algorithms introduced by Bryant.[31] Restrictions are imposed on the circuit description in order to achieve a canonical form for BDDs representing the circuit. This will make it possible to describe algorithms that reduce, merge, and otherwise manipulate BDDs. Given two combinational circuits represented in a reduced, ordered BDD canonical form, it becomes possible to compare the circuits in order to determine whether they represent different functions, or are just different expressions of the same circuit. The two circuits may originally be sum-of-products or product-of-sums, or one or both representations may be expressed at the RTL level. The canonical form also makes it possible to synthesize circuits described at different representations or levels of abstraction to the same resulting circuit.

The canonical form imposes a total ordering on the variables in a Boolean function of n variables. In this total ordering, the variables are numbered consecutively from 1 to n, and this numbering remains constant throughout processing. To achieve this ordering, it is convenient to simply label the variables as x_i, $1 \le i \le n$, as we have done previously. Vertices are assigned indices corresponding to the subscripts,

in ascending order. A graph formed in this fashion is called a *function graph*. Function graphs form a proper subset of conventional BDDs. By virtue of the numbering, the graphs are also acyclic.

Definition 2.6 A function graph G having root vertex v denotes a function f_v defined recursively as

1. If v is a terminal vertex:
   a. If *value* $(v) = 1$, then $f_v = 1$.
   b. If *value* $(v) = 0$, then $f_v = 0$.
2. If v is a nonterminal vertex with *index*$(v) = i$, then f_v is the function

$$f_v(x_1, ..., x_n) = \bar{x}_i \cdot f_{low(v)}(x_1, ..., x_n) + x_i \cdot f_{high(v)}(x_1, ..., x_n)$$

The formula for f_v is Shannon's expansion. A unique path from the root to a terminal vertex is defined by assigning logic values to all the x_i.

Definition 2.7 Function graphs G and G' are *isomorphic* if there exists a 1-to-1 mapping σ from the vertices of G onto the vertices of G' such that for vertices $v \in G$ and $v' \in G'$, either v and v' are both terminal vertices with value(v) = value(v'), or v and v' are both nonterminal vertices with index(v) = index(v'), σ(low(v)) = low(v'), and σ(high(v)) = high(v').

Proving that two function graphs are isomorphic begins by mapping the root of G onto the root of G'. The children of the root of G are then mapped onto the children of the root of G'. This mapping continues until either there are no more vertices to process, or an attempt to map a vertex in G to a vertex in G' fails.

Definition 2.8 For any vertex v in a function graph G, the *subgraph* rooted by v is defined as the graph consisting of v and all of its descendents.

Definition 2.9 A function graph G is *reduced* if it contains no vertex v with *low(v) = high(v)*, nor does it contain distinct vertices v and v' such that the subgraphs rooted by v and v' are isomorphic.

Theorem 2.5 For any Boolean function f, there is a unique (up to isomorphism) reduced function graph denoting f. Any other function graph denoting f contains more vertices.

The proof, by induction, can be found in Bryant's original paper. We now proceed to describe some algorithms introduced by Bryant. The most important of these algorithms are the *Reduce algorithm*, which transforms any arbitrary graph into a unique, reduced graph representing the same function, and the *Apply algorithm*, which performs a specified operation, such as AND, OR, XOR, and so on, upon two

BDDs. However, first it is helpful to define a data structure that describes the vertices in the BDD. The following structure, expressed in the C programming language, contains information needed to process the vertices, and facilitates traversals of the BDD:

```
struct vertex {
  struct vertex   *parent, *low, *high;
  int             index;
  int             id;
  char            value;  // 0, 1 or X
  char            mark;
}
```

Table 2.5 describes the entries in this structure. The index is taken from the subscript of variable x_i. The id field can be used when assigning numbers to the vertices during an operation. The *mark* field can be initially set to 0 or 1. Suppose the field is initially set to 0. Then, when traversing the BDD, *mark* can be set nonzero to indicate that the vertex has been visited. A simple rule when traversing the graph is to start at the root. Then, for vertex v, first visit $low(v)$ if it is unmarked. If it is marked, and if $high(v)$ is also marked, then set the *mark* of vertex v nonzero, and move up to the parent vertex. Repeat until all vertices are marked. This is described more formally in the following procedure:

```
procedure Traverse(v:vertex)
{
    v.mark := not v.mark;
//    ... perform operations here ...
    if (v.index < n+1)
    {       // v nonterminal
      if(v.mark != v.low.mark) Traverse(v.low);
      if(v.mark != v.high.mark) Traverse(v.high);
    }
    return;
}
```

TABLE 2.5 Field Values for BDD Structure

Field	Terminal	Nonterminal
low	null	$low(v)$
high	null	$high(v)$
index	$n + 1$	$index(v)$
val	$value(v)$	X

Traverse is a basic utility that is employed by other functions to perform tasks such as to search BDDs or to assign unique integers to each vertex that it visits. For example, a counter may be used and vertices assigned ids in either ascending or descending order.

It was previously stated that variables must adhere to a total ordering during processing. All operations performed on a BDD must adhere to that same ordering of the variables. If the order is changed, it must be changed for all operations. BDDs that adhere to this ordering are referred to as ordered BDDs (OBDDs). If, in addition to the ordering, the BDDs are reduced, using the Reduce algorithm, the OBDDs become reduced, ordered BDDs (ROBDDs). The ROBDDs produced by the Reduce algorithm are unique; hence if two circuits represented in BDD form, with their variables in the same order, are reduced to identical ROBDDs, then the original circuits from which they were derived are identical.

The Reduce algorithm is given below, in a pseudo-language. It will be illustrated using Figure 2.36. Note that it will be convenient to refer to a BDD representing function f as B_f. The first step is to group the vertices into $n + 1$ lists, where each vertex with index i is linked to list position i. This can be done using the Traverse algorithm. Then the linked lists are processed, beginning with list $n + 1$—that is, the list of terminal vertices.

```
function Reduce(v: vertex): vertex;
var subgraph: array[1..|G|] of vertex;
var vlist: array[1..n+1] of list;
{
  Put each vertex u in list vlist[u.index] // use
    // Traverse
  nextid = 0;
  for(i = n+1; i >= 0; i--); // start with terminal
    // vertices
  {
    Q = empty set;
    for(each u in vlist[i])do
      if (u.index == n+1)
        add <key,u> to Q // key=(u.value) (terminal
                         //   vertex)
      else if (u.low.id = u.high.id)
        u.id = u.low.id;     // redundant vertex
      else add <key,u> to Q; // key = (u.low.id,
                             //   u.high.id)
// NOTE: u.id not added to Q if (u.low.id == u.high.id)
    sort(Q);                 // by keys
    oldkey = (–1,–1);        // unmatchable key
    for(each <key,u> in Q) { //removed, in order
```

```
      if (key == oldkey)
        u.id = nextid;       // matches existing vertex
      else {                 // unique vertex
        nextid = nextid+1;
        u.id = nextid;
        subgraph[nextid] = u;
        u.low = subgraph[u.low.id];
        u.high = subgraph[u.high.id];
        oldkey = key;
      }
    }
    return(subgraph[u.id]);
  }
}
```

When processing the terminal vertices in vlist $n + 1$, a 2-tuple <key, u> is added to set Q for each terminal vertex $u \in$ vlist$[n + 1]$. Key is actually the value 0 or 1 of the terminal vertex. After all terminal vertices have been processed, the set Q is processed. Two terminal vertices are retained, one for each binary value. The terminal vertex with value 0 is assigned the id 1, and the terminal vertex with value 1 is assigned the id 2. These ids appear in enclosed in diamonds in Figure 2.36.

After the terminal vertices have been processed, the nonterminal vertices are processed, starting with vlist[n]. First, Q is reset to the empty set, and then each of the four vertices linked to vlist[3] is processed in turn. Note that for $i = n$, if u.low.id = u.high.id, then the low and high edges emanating from vertex u both terminate on a terminal vertex with value 0 or 1. Hence, the vertex can immediately be replaced by low(u). In Figure 2.36 the leftmost vertex with index 3 can be replaced by the terminal vertex with value 0. In practice, the low(v) from the leftmost vertex with index 2 can be connected to a terminal vertex with value 0.

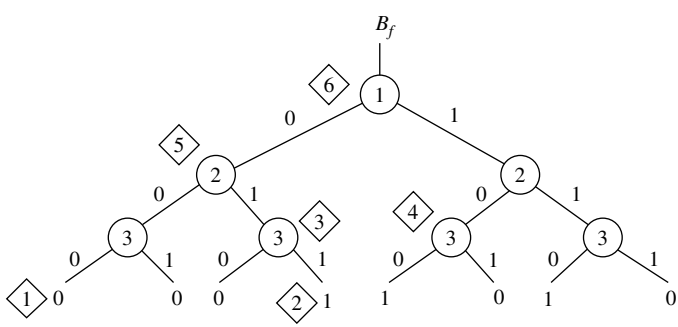


Figure 2.36 Assigning ids to vertices.

After the remaining vertices with index 3 have been processed, the set Q will have three entries corresponding to index 3. The first entry in Q will have key <1, 2>, and the remaining two entries will both have key <2, 1>. The keys are sorted and duplicates are discarded. In Figure 2.36, the rightmost vertex with index 3 is discarded and the 1-edge from its parent vertex is reset so as to point to the other vertex with key <2, 1>. The two remaining vertices with index 3 are assigned ids 3 and 4, again enclosed in diamonds.

The next vlist to be processed is vlist[$n - 1$], in this case vlist[2]. The leftmost vertex with index 2 is assigned key <1, 3>. The rightmost vertex with index 2 is discarded because its low(u) and high(u) both point to the same vertex. Hence, the 1-edge emanating from the root connects to the vertex with index 3 and id 4. The leftmost vertex is assigned id 5. Finally, vlist[1] is processed and assigned id 6. The ROBDD that results from applying the Reduce algorithm to the BDD in Figure 2.36 is shown in Figure 2.37. To build the equivalent ROBDD from the original BDD, it is necessary to keep track of the vertices in the ROBDD using linked lists. Then, after the entire original BDD has been processed, a ROBDD is constructed using the linked lists of vertices, adjusting pointers from discarded vertices to the vertices that were assigned ids. Finally, the original BDD can be discarded and its memory freed up.

It was stated earlier that variables must be ordered when creating ROBDDs. However, there is no rule dictating the order, only that the same ordering must be maintained during all processing. In fact, because ROBDDs are very sensitive to the ordering chosen, a considerable amount of research has been expended trying to find ideal orderings for the variables. For example, if the variables in Figure 2.37 are rearranged so that x_2 becomes the root, then the ROBDD in Figure 2.38 results. It represents the same function as the ROBDD in Figure 2.37, but has one more nonterminal vertex. Some functions are extremely sensitive to ordering of the variables.

2.11.3 The Apply Operation

Given two functions f and g, and a logic operation $\langle op \rangle$, the result $f \langle op \rangle g$ can be obtained by applying $\langle op \rangle$ directly to the expressions for f and g, using the distributive, commutative, and other familiar rules for manipulating Boolean expressions.

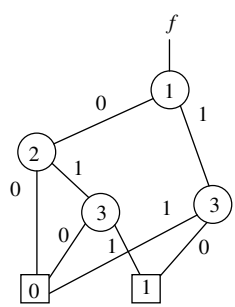


Figure 2.37 Reduced binary decision diagram.

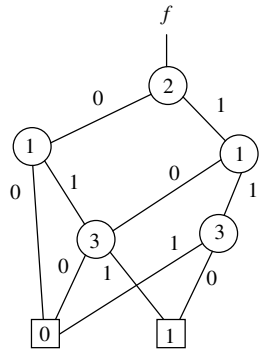


Figure 2.38 Another ROBDD for the same function.

Another approach is to apply $\langle op \rangle$ to the values of f and g in corresponding rows of their truth tables. A third method, given complete binary decision trees for f and g, is to apply $\langle op \rangle$ to corresponding terminal vertices of the trees. However, in practice, f and g are likely to be reduced, and available computer memory, in all likelihood, is not sufficient to permit expanding the OBDDs to binary decision trees. The Apply algorithm addresses this problem. Given two OBDDs B_f and B_g, Apply operates on them recursively and produces a resulting OBDD that represents $B_f \langle op \rangle B_g$. It is based on the following recursion, obtained by performing $\langle op \rangle$ on Shannon's expansion for the functions f and g:

$$f \langle op \rangle g = \bar{x}_i \cdot (f|_{x_i = 0} \langle op \rangle g|_{x_i = 0}) + x_i \cdot (f|_{x_i = 1} \langle op \rangle g|_{x_i = 1}) \qquad (2.12)$$

The Apply algorithm starts at the roots of two OBDDs B_f and B_g, corresponding to functions f and g, and descend toward the terminal vertices. At any time during the discussion that follows, the corresponding vertices of f and g that Apply is operating on will be considered roots r_f and r_g of corresponding subgraphs. The Apply algorithm is constantly producing resulting vertices $r_f \langle op \rangle r_g$. During this descent, there are several possibilities that must be considered:

1. Roots r_f and r_g are both terminal vertices.
2. Roots r_f and r_g are nonterminal vertices with identical indices i.
3. r_f is a nonterminal vertex with index i, and r_g is either a terminal vertex or a nonterminal with index j, for $j > i$.

If roots r_f and r_g are both terminal vertices, then the value of the terminal vertex for the resulting OBDD is value(r_f) $\langle op \rangle$ value(r_g). If roots r_f and r_g are nonterminal vertices and have identical indices i, then the Apply algorithm is applied to the low and high vertices of r_f and r_g; that is, the corresponding vertex of the resultant

OBDD has a 0-arc to apply(⟨*op*⟩, low(r_f), low(r_g)) and a 1-arc to apply(⟨*op*⟩, high(r_f), high(r_g)). This is basically an iteration of Shannon's equation, as expressed in Eq. (2.12). The third case requires a little more analysis. Note that there is actually a fourth case where $i > j$, and r_f is either nonterminal or terminal. However, the problem is symmetrical, so the processing follows that of case 3.

If the root r_g has index $j > i$, then the subfunction corresponding to r_g is independent of the variable x_i. In that case, $g|_{x_i = 0} = g|_{x_i = 1}$. So

$$g = \bar{x}_i \cdot g|_{x_i = 0} + x_i \cdot g|_{x_i = 1} = (\bar{x}_i + x_i) \cdot g|_{x_i = 0} = g|_{x_i = 0}$$

Therefore $g|_{x_i = 0}$ and $g|_{x_i = 1}$ in Eq. (2.12) can both be replaced by g. As a result, the 0-arc in the resultant OBDD is determined by apply(⟨*op*⟩, low(r_f), r_g) and the 1-arc is determined by apply(⟨*op*⟩, high(r_f), r_g). If r_g is a terminal vertex, then ⟨*op*⟩ may cause the resulting vertex to assume a binary value, in which case the resulting vertex is terminal. This would happen, as an example, if r_g is terminal with binary value 0 and ⟨*op*⟩ is an AND operation.

The Apply algorithm follows:

```
function Apply(v1, v2: vertex; <op>: operator): vertex;
  var T: array[1..|G1|, 1..|G2|] of vertex;
  function Apply-step(v1, v2: vertex): vertex;
                 // recursive
  {
    u = T[v1.id, v2.id];
    if (u != NULL)
      return(u);              // already evaluated
    u = new vertex record;
    u.mark = FALSE;
    T[v1.id, v2.id] = u;     // add vertex to table
    u.value = v1.value <op> v2.value;
    if (u.value != X) {      // create terminal vertex
      u.index = n+1;
      u.low   = NULL;
      u.high  = NULL;
    }
    else {  // create nonterminal, continue descent
      u.index = Min(v1.index, v2.index);
  //-----------------------------------------------
      if (v1.index == u.index)
           { vlow1 = v1.low; vhigh1 = v1.high; }
      else { vlow1 = v1;     vhigh1 = v1;      }
```

```
  //-------------------------------------------
      if (v2.index == u.index)
          { vlow2 = v2.low; vhigh2 = v2.high; }
      else{ vlow2 = v2;     vhigh2 = v2;      }
  //-------------------------------------------
      u.low  = Apply-step(vlow1,  vlow2);
      u.high = Apply-step(vhigh1, vhigh2);
    }
    return(u);
  }
{     // Main routine
  initialize all elements of T to null;
  u = Apply-step(v1,v2);
  return(Reduce(u));
}
```

The Apply algorithm will be illustrated using the circuit in Figure 2.39. The OBDDs B_f and B_g represent the AND gates f and g. All 0-arcs go directly to terminal vertex with value 0. The object will be to synthesize an OBDD for the entire circuit, given the OBDDs for f and g.

The B_f in Figure 2.40 is an expanded version of the B_f in Figure 2.39. In Figure 2.40 there are two vertices with index 2. Both edges terminate on a vertex with index 3. Likewise, the vertex with index 3 has two edges terminating on the terminal vertex with value 0. It would be possible to completely expand a BDD to achieve a binary decision tree—that is, one in which all possible terminal vertices exist. Then a logic operation could be applied to corresponding terminal vertices. However, Apply does not pad the BDD in this way. Rather, if one BDD has a vertex

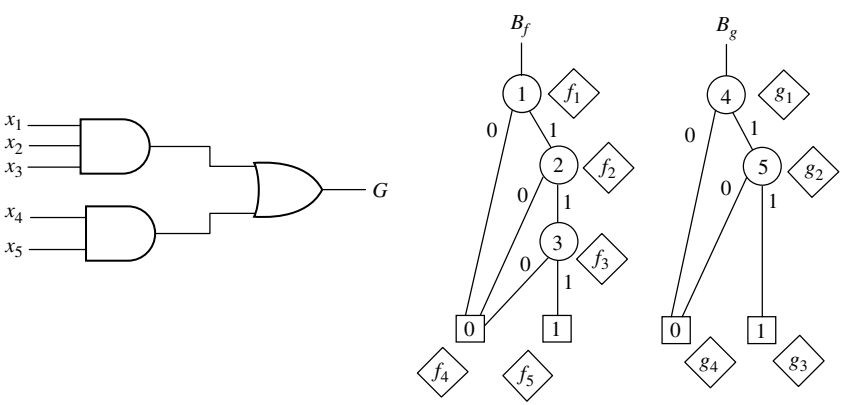


Figure 2.39 OR'ing two BDDs.

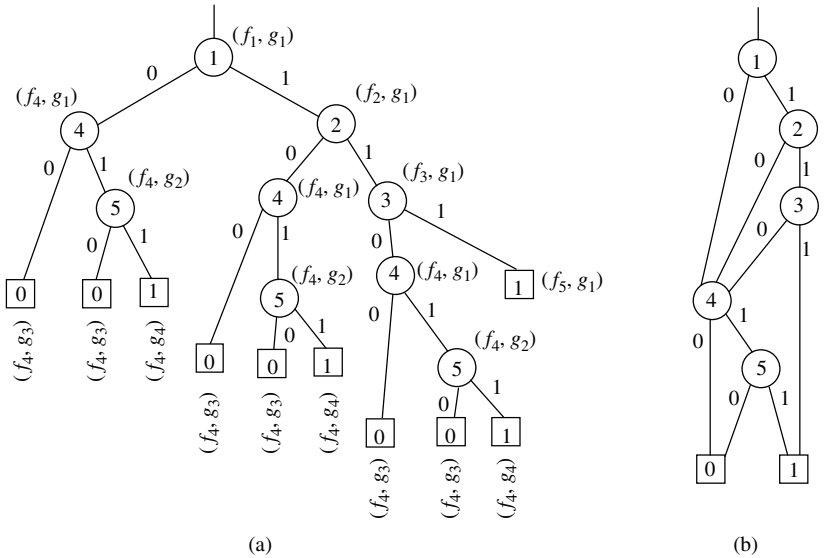


Figure 2.40 Applying the Apply algorithm.

at position i and the other does not, then Apply goes directly to the vertex at position j, where $j > i$. If $j = n + 1$, then performing $\langle op \rangle$ on a pair of vertices may cause a terminal vertex to be created. For example, if $\langle op \rangle$ is the AND operation, and one vertex is a terminal vertex with value 0, then performing $\langle op \rangle$ on that vertex and any other vertex from the other BDD will always result in a terminal vertex with value 0.

The Apply algorithm will be illustrated by OR'ing ROBDDs B_f and B_g in Figure 2.39. The calculations are shown in Figure 2.40(a), and the reduced ROBDD is shown in Figure 2.40(b). The starting point for the Apply algorithm is the pair of root vertices, f_1 and g_1. The first step is to create a root vertex corresponding to the OR of B_f and B_g. In Figure 2.40(a) this vertex is assigned the label (f_1, g_1). From there, Apply begins its descent down the edges of each OBDD. It first calculates low(f_1, g_1). Starting at the low edge of f_1, it arrives at terminal vertex f_4, with index(f_4) = 6. Since index(g_1) = 4, which is less than index(f_4), Apply remains at g_1. The OR operation is applied to terminal vertex f_4 and nonterminal vertex g_1, and it yields vertex g_1.

Apply then calculates high(f_1, g_1). Index(high(f_1)) = 2 and index(high(g_1)) = 5, so Apply stays at g_1, rather than descending to its child vertex. The OR applied to f_2 and g_1 is indeterminate, so a nonterminal vertex with index 2 is created and assigned the label (f_2, g_1). Next, Apply processes vertices f_4 and g_1. Low(f_4) and low(g_1) are both terminal vertices with values 0, so performing the OR operation on these vertices results in a terminal vertex that is assigned the label (f_4, g_3). Processing high(f_4) and high(g_1) produces a vertex with label (f_4, g_2) and index 5. The remaining vertices are processed in similar fashion.

Note that in Figure 2.40(a) some vertices appear more than once. For example, vertex (f_4, g_1) appears three times. The subgraph with root (f_4, g_1) need not be processed each time it is encountered. The table T is used to identify vertices in the resultant BDD that have already been processed. When such a vertex is encountered, a pointer to the original vertex is inserted in the BDD. This can result in significant savings in processing time. Because T may represent a sparse matrix, the actual implementation can be a hash table in order to minimize the amount of memory required.

The *Restriction* algorithm is a useful utility. Given a function f, Restriction converts f into $f|_{x_i = b}$. Restriction traverses the BDD, like Traverse, looking for pointers to a vertex v such that index(v) = i. When such a pointer is encountered, it is changed to point to low(v) if $b = 0$, or it is changed to point to high(v) if $b = 1$. Then Reduce is called to reduce the graph.

The *Composition* algorithm is used to obtain a graph for a hierarchical network. For example, an n-wide adder may contain n full adders connected in a ripple carry configuration. The following equation represents a function f_1 for which function f_2 is to be substituted for variable x_i. The ROBDD for this function can be derived directly through application of the Restriction and Composition algorithms, followed by Reduce. A more efficient implementation of the Composition algorithm can be found in Bryant's original paper.[31]

$$f_1|_{x_i = f_2} = \bar{f}_2 \cdot (f_1|_{x_i = 0}) + f_2 \cdot (f_1|_{x_i = 1})$$

2.12 CYCLE SIMULATION

New design starts continue to grow in gate count, and the amount of CPU time required to simulate these designs tends to grow disproportionate to gate count, implying a growing need for simulation speed. A simple example helps to shed light on this situation. Suppose a circuit has n functions and that, in the worst case, each function interacts with all of the others. Ignoring for the moment the complexity of the interactions, there are $n \times (n - 1)/2$ potential interactions between the n functions. Thus, in the worst case, the number of interactions grows proportional to the square of the number of functions.

Handshaking protocols between functions also grow more complex. Internal status and mode control registers act as extensions to device I/O pins. To verify the growing number of interactions requires more stimuli. In addition, the growing number of gates and functions in the circuit model generate more events that must be evaluated during each clock cycle. The combination of more functionality and more stimuli requires an exponentially growing amount of CPU time to complete the evaluations. A consequence of this is a growing difficulty to create and simulate enough stimuli to verify design correctness. As a result, design errors are more likely to escape detection until after tape-out, at which time the discovery of errors requires another expensive iteration through the design cycle.

Cycle simulation is one of the answers to the growing need for greater verification power. *Cycle simulation* evaluates logic elements and functions across clock cycle boundaries without regard to intermediate values. Its purpose is to evaluate input stimuli as rapidly as possible. Designs are required to be synchronous so that every possible technique can be leveraged during simulation. Rank-ordering is used so that elements only need to be evaluated once during each clock period. Circuit delays are ignored, and the number of logic values is usually limited to three or four {0, 1, X, Z}. Internal representation of the circuit may be in terms of binary decision diagrams (BDDs), so intermediate values are totally obscured. To insure that a circuit operates at its intended speed when fabricated, circuit delays are measured by timing analysis programs that are written specifically for that purpose and run independently of simulation. The designer plays a role in this simulation mode by modeling circuits at the highest possible level of abstraction without losing essential details.

A number of methods have been developed to speed up simulation while reducing the amount of workstation memory required to perform simulations. Figure 2.41 provides a taxonomy of such approaches.[32] From the figure it can be seen that simulation performance can benefit from enhancements in software, hardware, and circuit modeling. Chapter 12 will examine analytical methods for design verification.

Modeling efficiencies can be realized in several ways. The Verilog HDL supports user defined primitives (UDPs). These permit a user to define the behavior of small functions such as multiplexers, full-adders, latches, delay flip-flops, and so on, by means of lookup tables rather than as interconnections of several individual logic gates. A single table lookup then replaces several logic gate evaluations.

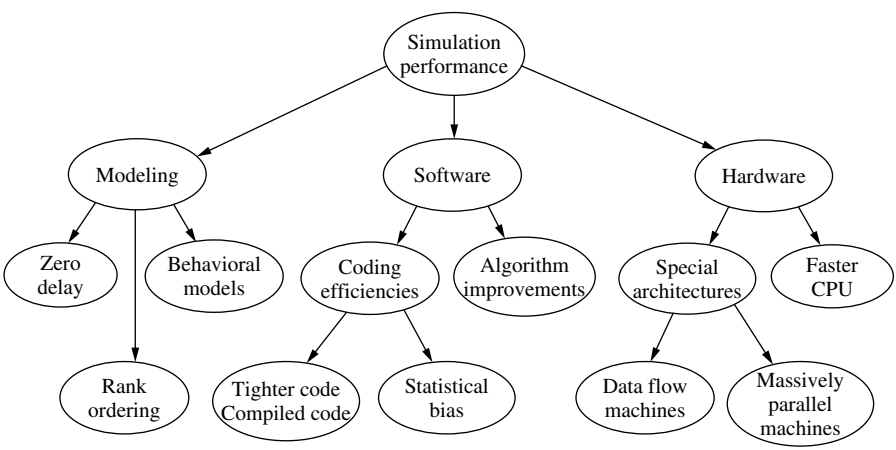


Figure 2.41 Simulation performance factors.

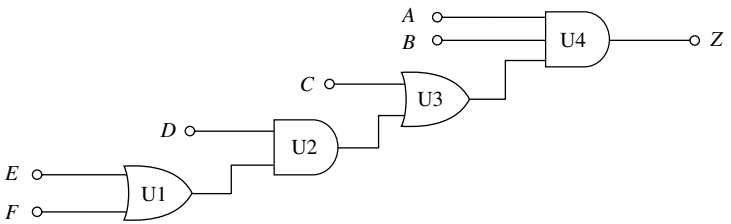


Figure 2.42 Computing output value efficiently.

Statistical bias can be used to advantage both in the simulator and in the model. Consider the circuit in Figure 2.42. In Verilog the circuit might be coded as

$$Z = A \;\&\; B \;\&\; (C \mid (D \;\&\; (E \mid F)));$$

An intelligent simulator will process it as if it had been encoded as

```
if ((A == 0) | (B == 0)) Z = 0;
else if (C == 1) Z = 1;
else if (D == 0) Z = 0;
else if ((E == 1) | (F == 1)) Z = 1;
else Z = 0;
```

As soon as the value of Z has been determined, the simulator breaks out of the if/else construct since there is no need for further processing. If logic values 0 and 1 are equally probable on all nets, then 50% of the time A is 0 and further calculations cease. Similar considerations hold for B, so that 75% of the time it is unnecessary to go beyond the first line. Similar considerations hold for the remaining lines.

Rank-ordering was discussed in Section 2.6, where it was pointed out that it was a necessary requirement for efficient simulation. An event-driven simulator does not require rank-ordering to correctly simulate a circuit, but can benefit from it. If a combinational array such as an ALU or multiplier is being evaluated, rank-ordering can ensure that no element is evaluated more than once. However, either all elements must be assigned zero delay or, if delay values are present, they must be ignored. The simulator can be implemented with both the timing wheel and the READ/WRITE array scheduling mechanisms. Then, the more efficient READ/WRITE array can be used in place of the timing wheel when groups of zero-delay logic are encountered in order to realize further CPU savings. In general, the use of two scheduling mechanisms permits synchronous and asynchronous logic to be segragated and processed separately.

Stimulus ordering refers to the practice of ordering stimuli at primary inputs in such a way as to reduce the number of logic events propagating through a circuit. When simulating a combinational circuit where simulation results do not depend

on the existing state of the circuit, a common practice is to apply randomly generated stimuli to the circuit to verify its correctness. Large numbers of vectors can be generated with very little effort on the part of the person performing the verification. For example, if verifying an array multiplier, the logic designer can write a computer program to randomly generate input arguments *A* and *B* as integers, multiply them to obtain the product, then decompose *A* and *B* into their binary equivalents and apply them to the design. The binary result computed during simulation is then converted to decimal and compared with the value computed by the computer program.

When many random input values change from one vector to the next, a huge number of simulation events can occur in a gate-level circuit model. On large combinational arrays with thousands, or tens of thousands, of logic gates, ordering vectors based on their Hamming distances (cf. Chapter 10) can sometimes produce major savings of simulation time. To understand the principle, consider a simple 2-input AND gate. If the input combinations are ordered as *A,B* = {(0,0), (1,1), (1,0), (0,1)}, there are a total of five input events. If the input combinations are reordered as *A,B* = {(0,0), (0,1), (1,1), (1,0)}, each vector causes a single input event, so there are a total of three input events. For a combinational block of logic, results are not affected by the order in which vectors are simulated, so rearranging the input vectors in order to minimize events from one vector to the next may yield significant savings in CPU time.

In general, the goal of *cycle-based simulation* is to squeeze out all unnecessary computations while correctly determining circuit response to input stimuli. In order to eliminate computations, assumptions usually must be made. For example, it must be safe to assume that hazards will not destabilize the circuit. To safely make this assumption, state transitions must be synchronized by external clock(s) that are unaffected by internal logic activity. Furthermore, the durations of clock periods must be independent of circuit activity, and it is necessary to verify, independent of simulation, that logic events in the circuit will propagate to their destinations within the allotted time period.

If a circuit can be correctly simulated with only the values 0 and 1, the circuit model can be further simplified, and control statements, such as *case* statements and *if* statements, do not have to consider the consequences of indeterminate values. But, to get correct values, it must be possible to initialize all flip-flops to 1 or 0 at the beginning of simulation. Storage elements must be explicitly defined. This means that storage created by feedback loops in combinational logic, such as latches created by cross-coupled NAND or cross-coupled NOR gates, must be forbidden.

Wherever possible, blocks of detailed circuitry should be replaced by models expressed at a higher level of abstraction, eliminating intermediate variables along the way. If, for example, an ALU has been thoroughly characterized and its behavior can be expressed by a case statement, that code should be used in place of a more detailed RTL or gate-level model. This is especially true when running regression tests, provided that the circuitry expressed at a higher level of abstraction has not, itself, become the subject of change activity. The circuit in Figure 2.43 can be used

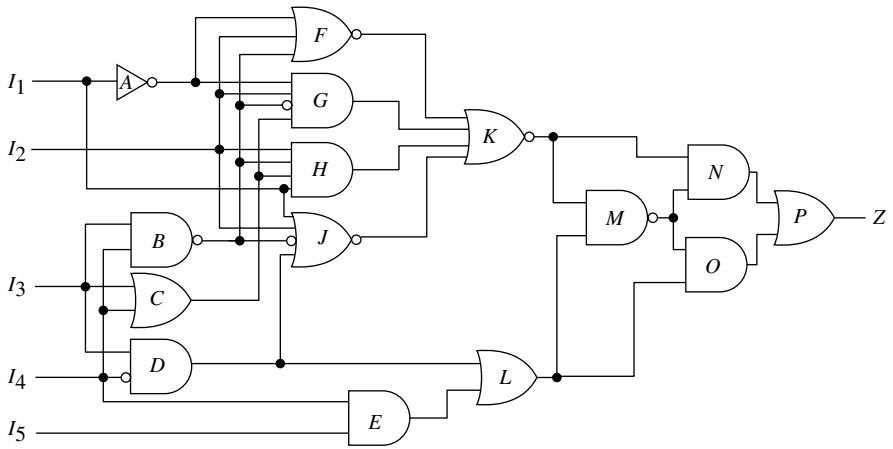


Figure 2.43 Circuit illustrating cycle simulation.

to illustrate this. A more concise description of its behavior is provided by the following Verilog code:

```
module litl_alu (i1,i2,i3,i4,i5,z);
input i1, i2, i3, i4, i5;
output z;
reg z;
always @(i1 or i2 or i3 or i4 or i5)
  case({i3,i4})
  2'b00: z = i1 | i2;
  2'b01: z = i1 ^ i2 ^ i5;
  2'b10: z = i1 & i2;
  2'b11: z = !(i1 ^ i2 ^ i5);
  endcase
endmodule
```

The use of ROBDDs to evaluate cones of logic can provide huge performance gains. Consider first the evaluation of the circuit using a zero-delay simulator. All the nets are initialized to X, and then the vector I_1, I_2, I_3, I_4, $I_5 = (0, 0, 0, 0, 0)$ is applied to the circuit. Every element in the circuit has to be evaluated. Now suppose I_2 switches to 1. Gates J, K, N, and P switch states. Each logic gate evaluation requires that the simulator acquire two or more values corresponding to the inputs of that gate and perform the appropriate calculation. The evaluation of the RTL code significantly reduces the amount of computation required.

Now consider what happens when ROBDDs are used. The ROBDD for the circuit in Figure 2.43 is shown in Figure 2.44. To determine the output response of the

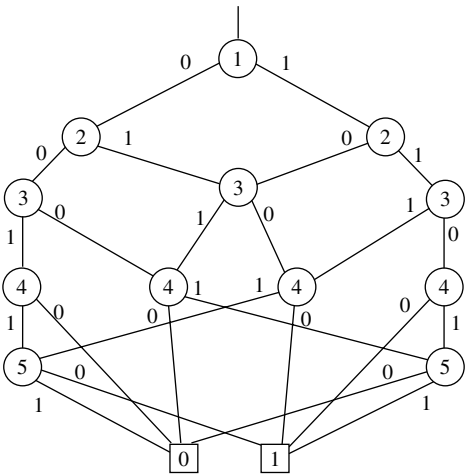


Figure 2.44 ROBDD for circuit in previous figure.

circuit for the input combination (0,0,0,0,0), simply traverse all the 0-arcs of the ROBDD. Recall from the previous section that there is a data structure for each vertex, and the data structure contains pointers corresponding to the 0-edge and the 1-edge. It is a simple matter to traverse these structures until arriving at a terminal vertex, in this case the vertex with value 0. When I_2 changes to 1, the entire ROBDD is again traversed; however, this time the path leads to the terminal vertex with value 1.

In both traversals it was only necessary to follow links in data structures corresponding to four vertices. For a larger combinatorial array, such as an ALU, the savings in CPU time may be two or more orders of magnitude. The one drawback to this approach is that BDDs for some arrays, such as multipliers, cannot be reduced. When circuits contain large arrays whose BDD representation cannot be reduced and are too large to fit into memory, a hybrid approach can be used. Those networks can be rank-ordered and simulated using event propagation. Other judgments can also be made; for example, if an RTL expression is obviously a counter, then the entire block of code representing the counter can be treated as a single function and simulated as such. This will require that the logic designer model constructs such as counters unambiguously, so the simulator can recognize their behavior.

2.13 TIMING VERIFICATION

As systems grow larger and as design, simulation, and test grow more complex, synchronous design techniques become more attractive. The use of one or more master clocks to synchronize events makes it possible to simulate logical and functional behavior in a zero delay environment. If, in addition, the system is provided with a master reset that forces all memory elements into a known starting state, it becomes

possible to dispense with the indeterminate X value and restrict simulation to the Boolean values 0 and 1.

A key feature of this design methodology is the fact that *all* registers and flip-flops are controlled by one or more clock signals that are either not gated with combinational logic or are gated only within the framework of a very closely controlled set of design rules. This operation is illustrated in Figure 2.45 for a circuit with a single clock. The elements labeled *A, B, C*, and *D* may be registers or single flip-flops. At no time in this circuit is any clock signal generated or controlled by logic operations performed in combinational logic. Clock line layout, powering, and delay calculations are performed independently of the logic controlled by the clocks.

Just as clock distribution is a science independent of logic design, zero-delay simulation requires an independent means for computing propagation delay along signal paths. If delay is excessive, a signal will not reach its destination before the next clock pulse. If the delay is too short, hold time requirements for the flip-flops may be violated. Two methods for performing timing verification include path enumeration and block oriented analysis.[33]

2.13.1 Path Enumeration

Path enumeration starts at a particular element, either an I/O pin or a stored state variable, and traces through the logic until a termination point is reached, either an I/O pin or a stored state variable. Maximum element delays encountered along the paths are added to accumulative a total as the program traces the path. Rise and fall times are both used to precisely calculate propagation time.[34]

Example The circuit in Figure 2.46 will be used to illustrate path enumeration. To calculate the propagation time required for a signal originating at *E* to reach *L*, start at *L* and work back toward the inputs. Assume that a rising signal has reached *L*. In that case the rise time for gate *K* is used as the initial sum. It is added to the rise time for gates *I* and *J*. The fall time for *G* is added next because a 0 to 1 transition at the output of gate *J* requires a 1 to 0 transition at input *E*. Next, the propagation time for a falling signal to reach gate *L* is calculated. To get this value the fall times for gates *K, I*, and *J* and the rise time for gate *G* are added. The larger of the two sums becomes the propagation time from *E* to *L*. ■■

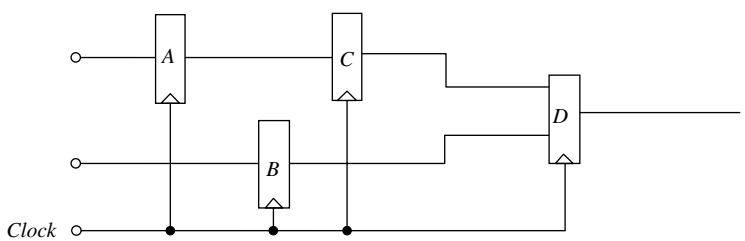


Figure 2.45 Synchronous circuit.

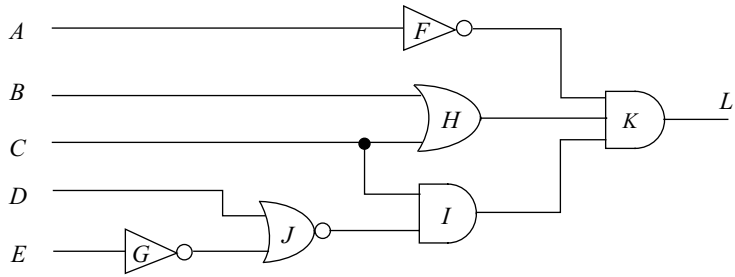


Figure 2.46 Path analysis circuit.

An important point in the rationale for timing verification is the fact that, at some point during operation of a circuit, the signal along the path being calculated will be the controlling signal for some output. For example, if inputs *A,B,C* and *D* in Figure 2.46 are assigned the values (0,0,1,0), then the output is totally dependent on the value assigned to input *E*. If it has value 0(1), then output *L* has value 0(1). When the path being analyzed is the controlling signal, path enumeration must determine which signal originating at the input, 0 or 1, takes longer to propagate to the output. It must then determine, among all paths into a bistable, the path that has maximum propagation delay when it has the controlling signal. The implicit assumption that all other signals are set up to propagate the signal whose delay is being calculated makes it possible to ignore the logic function performed by the elements along that path. It is only necessary to know the rise and fall delays of each element and whether or not the element inverts the signal.

2.13.2 Block-Oriented Analysis

In this method the program starts at some assumed time with signals at primary inputs and bistables. Furthermore, required arrival times are assigned to destination elements. The elements, or blocks, that are driven by the primary inputs and bistables are processed to find the earliest and latest time at which a signal could propagate through them. Then, elements driven by these elements are processed. In general, no element is processed until all elements driving its inputs are processed. This requires that the circuit be rank-ordered.

The block-oriented method identifies the worst path leading up to each block and feeds this information forward. This is continued until a primary output or bistable is reached. Then, the difference between the required arrival time and the propagation time is computed. This value is called *slack*. A negative slack indicates excessive propagation time.

After all paths have been propagated forward, computations are performed in the opposite direction. The propagation value at the element that drives the primary output or bistable is subtracted from the required arrival time to determine when the signals must arrive at the inputs to this block. The previously computed propagation

numbers are subtracted to find the slack at the inputs to this block, and the process is continued until the source elements are reached.

Example Referring again to Figure 2.46, assume each of the elements has identical rise and fall delay of 5 units. Also, assume that input changes occur at time 0 and that maximum propagation delay to output L is 18 units. Gates F and H can both be processed to give delay of 5 units on their outputs, but J cannot be processed until G is processed. After G is processed, the delay at the output of J is the greater of the values on D and G plus the delay of J. Since the delay at G is 5 units, the delay at J is 10 units. In similar fashion, the delay at I is 15 units and the delay at primary output L is 20 units, which results in a slack of −2 at the output.

The computations are now performed in reverse, starting with the required arrival time and using the previously calculated propagation times. The slack on the inputs to K are +8, +8, and −2, derived by computing the required arrival time at the inputs to K, $18 - 5 = 13$, and subtracting from that the propagation delay at the outputs of F, H, and I. The required arrival time at the inputs to F, H, and I is $13 - 5 = 8$. The slack at the inputs to F and H is 8 and the slack at the inputs to I are +8 and −2. Continuing, we find that the slack at E is −2 and a critical path with excessive propagation time has been identified. ■■

If looking for early arrival times, the computations use minimum values. If separate rise and fall times are used, then pairs of numbers are maintained and inverting elements must be identified. A falling edge delay at the output of an inverting element is computed by taking the greater of the rise delays at its input and adding the fall delay of the element.

The object of timing verification is to find signal paths having long (or short) delay times. If propagation time along such paths is excessive, the path delay can be reduced either by redesigning the logic, by selecting faster components, or by assigning different physical dimensions to elements within an IC. A consequence of redesigning circuits to switch faster is that they may then consume more power. Increased power consumption may be offset by finding signal paths where the timing margin is greater than it needs to be and, if possible, redesigning the devices to consume less power.[35]

A major benefit of timing verification is the fact that signal paths do not get overlooked. Simulation only provides information on those signal paths that are exercised by the applied stimuli. By contrast, during timing verification all paths are (or can be) analyzed. However, some practical considerations must be taken into account. Path enumeration can generate large amounts of data. It may be necessary to reduce the amount of data generated so that the user is not overwhelmed. To achieve this, it must be possible for the user to specify printout only of paths that fall within some user-defined range, either above or below some threshold value.

For engineering design changes, it is not necessary to recompute all paths; therefore the user should have an option to specify signal paths of interest. Other considerations include the ability to detect and properly handle feedback paths in combinational logic, as well as paths that exceed some given clock period but which

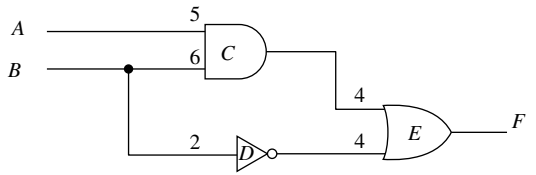


Figure 2.47 A false path.

are known to require two or more clock cycles to complete their operation. Clock skew must be factored into the overall analysis since the time required for a clock signal to reach numerous devices throughout a design, whether a chip or board, can vary significantly.

The user may have to be careful to spot paths that appear to be problem paths but which require logic combinations that cannot occur in practice. An example of this is redundancies in combinational logic. Consider the circuit in Figure 2.47. The delays are indicated at the inputs to the logic elements, and the rise and fall delays are assumed to be identical. The total delay from input A to output F is 9 units. From B to F through C is 10 units and from B to F through D is 6 units. It would appear that the longest delay path from any input to output F is 10 units. But, closer examination of the circuit reveals that it implements the function $A \cdot B + \overline{B}$, which can be simplified to $A + \overline{B}$, so the apparent longest path is redundant. This is an example of a *false path*.

2.14 SUMMARY

Simulation techniques span the spectrum from switch-level to behavioral. At one end of the spectrum, switch-level simulation provides considerable detail about the behavior of virtually every transistor in the circuit. However, there is a price to pay for this detail. Simulation takes much longer to complete. At the other end of the spectrum, behavioral simulation provides very little detail. It is not concerned with how the response is computed; its purpose is to investigate architectural parameters and trade-offs. RTL and gate-level simulation lie somewhere in the middle of this spectrum. The object at these levels is to design a circuit at the highest possible level of abstraction that can be processed by synthesis tools. Nevertheless, there are occasions, particularly with commodity chips, when design at the transistor level, at least for part of the chip, may be necessary in order to meet performance goals or die size restrictions.

The two basic approaches to simulation are interpreted and compiled. Interpreted simulation does not require preprocessing circuits into machine language models. For short simulation runs, an interpretive simulation may operate more efficiently, since the compiled simulator has greater overhead when creating the model. A compiled simulation executes more efficiently once the circuit is compiled. Hence for

simulation jobs where large amounts of stimuli are to be applied, such as regression suites that are run frequently, compiled simulation may be the preferred mode of operation.

An understanding of the concepts underlying simulation, at its various levels of abstraction, benefits users as well as those who implement the tools. By understanding the concepts involved, including the cost/benefit trade-offs, the user can select the right tool for his or her application. In future chapters we will see that this is true of other aspects of test, including fault simulation and ATPG. A word of caution is in order about abstraction. The process of abstraction strips away irrelevant detail in order to focus on parameters of interest. Determining which detail is relevant and which is irrelevant requires some judgment and experience. As an example, zero-delay simulation runs faster than nominal-delay simulation, but if applied to an asynchronous design, simulation results may become totally meaningless.

When dealing with digital circuits, large numbers of value/strength symbols may seem unusual to the inexperienced logic designer. We are accustomed to thinking in terms of 1s and 0s. Nevertheless, this spectrum of values has proven its worth. One of the early architects of a family of computers has explained to this author how a persistent problem in one of the models was traced to an uninitialized node. A new simulator, which incorporated the value U, representing uninitialized, was employed after the model had been in service for six months, and it successfully identified the troublesome node. On yet another occasion, a noisy bus caused reliability problems. An interim solution was the use of a piece of wire acting as an antenna. When noise became excessive, the clock was shut down. Eventually, with the help of simulation, the noise problems were tracked down and resolved.

Simulation technology has made great strides in the past three decades, both in terms of simulation speed and gate count of the circuits processed. Users have become more sophisticated in their choice of simulator algorithm, using switch-level where necessary, and behavioral simulation, sometimes aided by hardware accelerators, where possible. Advances over the past decade in simulation technology have been aided by the emergence and growing popularity of two hardware design languages, Verilog and VHDL. Successive generations of these languages are approaching a common base.

PROBLEMS

2.1 Prove that $A \cdot B + C \cdot D = (A + C) \cdot (B + C) \cdot (A + D) \cdot (B + D)$.

2.2 Design a JK flip-flop based on the D flip-flop.

2.3 Modify the compiled simulator of Section 2.6 to enable it to perform three-valued simulation on the cone of logic in Figure 2.9.

2.4 Modify the compiled simulator of the previous problem so that it can perform 3-valued simulation on a cross-coupled NAND latch. Create pseudo-inputs and pseudo-outputs, check for oscillations.

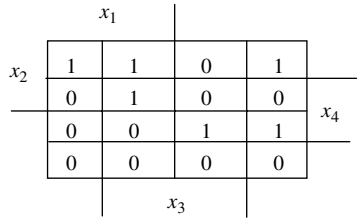


Figure 2.48 Karnaugh map.

2.5 State a general rule determining the minimum duration necessary for the pulse on the *Enable* line of the circuit in Figure 2.8(b) in order to prevent a glitch.

2.6 For the Karnaugh map in Figure 2.48:

(a) Identify a 1-hazard.

(b) Identify all transitions for which 1-hazards can be avoided.

(c) Find a dynamic hazard.

2.7 Using a Karnaugh map, explain why the hazard in the circuit of Figure 2.11 is prevented by the additional AND gate.

2.8 Assume that the buffers in Figure 2.49 have delays indicated by the numbers following the pound signs, and assume that all gates have zero delay. Also assume a signal change from $A,B,C,D,E = (0,1,1,1,0)$ to $A,B,C,D,E = (1,0,0,0,1)$ occurs. How many evaluations are required by an event-driven simulator to determine the state of the circuit? Count each event propagation through the delay elements as one evaluation. Next, assume that the buffers have zero delay and that the circuit is rank-ordered. How many evaluations are required under those assumptions?

2.9 In Figure 2.50, if elements are evaluated starting with the event occurring at input A_1, and then in ascending order to input A_n, how many events must be propagated? If the elements are evaluated in descending order, from input A_n to input A_1, how many events must be propagated?

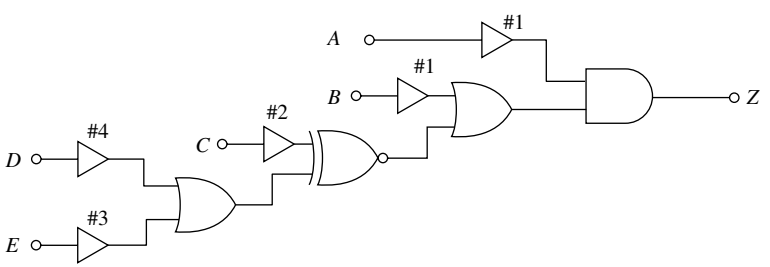


Figure 2.49 Delay calculations.

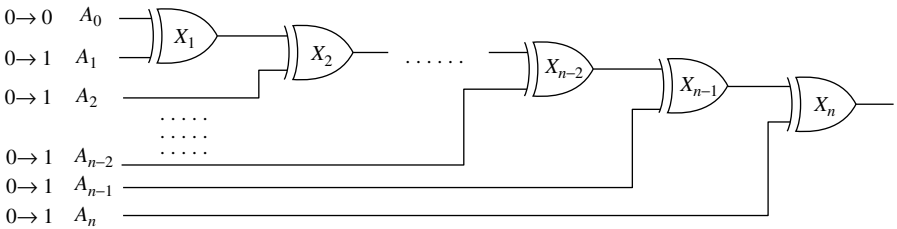


Figure 2.50 Event propagation.

2.10 Rank-order the circuit in Figure 2.43 and assign level numbers to each of the gates.

2.11 Using the delay flip-flop in Figure 2.7, cut the feedback lines and explain how to perform a zero-delay simulation, using Procedures A and B of Section 2.6.5. Apply the following sequence of inputs: $\overline{Preset}$, *Clock, Data,* $\overline{Clear}$ = {(1,0,1,0), (1,0,1,1), (1,1,1,1), (1,0,0,1), (1,1,0,1), (0,1,0,1)}. Show details of your work.

2.12 Using the delay flip-flop in Figure 2.7, assume that the rise and fall propagation times of the NAND gates are all 5 ns. What happens when an active clock edge appears with a pulse width of 8 ns? What is the minimum required setup time required for the circuit? What is the minimum required hold time?

2.13 Consider the circuit in Figure 2.51. Assume the initial assignment of values on the nodes are all Xs and that the circuit is rank-ordered; that is, no element is evaluated until all its inputs have been evaluated. Assume the input values are applied in ascending order; that is, *A,B,C,D* = {(0,0,0,0), (0,0,0,1), ..., (1,1,1,1)}. How many evaluations are necessary to complete the simulation? Suppose inputs are reordered as follows: *A,B,C,D* = {(0,0,0,0), (1,1,1,1), (0,0,0,1), (1,1,1,0), ..., (0,1,1,1), (1,0,0,0)}. Now how many evaluations are necessary? Find a stimulus ordering that minimizes the number of calculations required to simulate all 16 input combinations.

2.14 Create a nine-valued simulation table capable of detecting hazards at an OR gate.

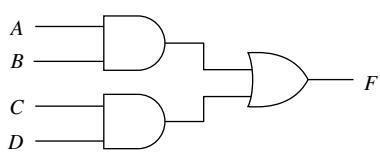


Figure 2.51 Counting events.

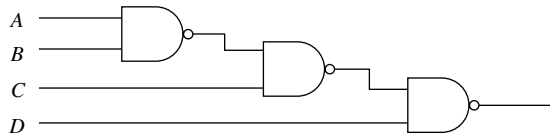


Figure 2.52 Path timing.

2.15 Given the following four combinations on the inputs of a three-input AND gate, what is the resulting output for each of the combinations?

input 1	M	/	W	M
input 2	W	^	^	*
input 3	1	\	^	M

2.16 Prove Lemmas 2.1 and 2.2 and Theorems 2.1 through 2.4.

2.17 Using Figure 2.52:

(a) Compute the timing of the paths from A, B, C, and D to the output for both 1 and 0. Assume the rise time of the NAND gates is 8 ns and the fall time is 5 ns.

(b) What maximum value would you get if you ignored the signal inversions and just used average propagation delay? Maximum propagation delay?

2.18 Referring to the circuit in Figure 2.29, describe the events that take place when inputs I_1 and I_2 change from (0,0) to (0,1), then to (1,0), and then to (1,1). What is the function of that circuit? Describe it in terms of Verilog PMOS and NMOS transistors. Describe it in terms of tranif0 and tranif1 transistors.

2.19 Partition the circuit in Figure 2.29 dynamically and evaluate the circuit for the four input combinations. Show your calculations.

2.20 Partition the circuit in Figure 2.26(b) statically. Describe the events that occur when the cell has value 0 and is being updated to store a logic 1.

2.21 In the example using Figure 2.30, change input B from 1 to X and recompute the node and switch values.

2.22 Are the two circuits in Figures 2.53(a) and 2.53(b) equivalent? Explain your answer.

2.23 Partition the circuit in Figure 2.54 into components. Apply various binary combinations to inputs A, B, C to determine the function of the circuit.

2.24 Using the gate-level model in Figure 2.43, the RTL model (litl_alu), and the ROBDD in Figure 2.44, contrast the amount of work that must be performed

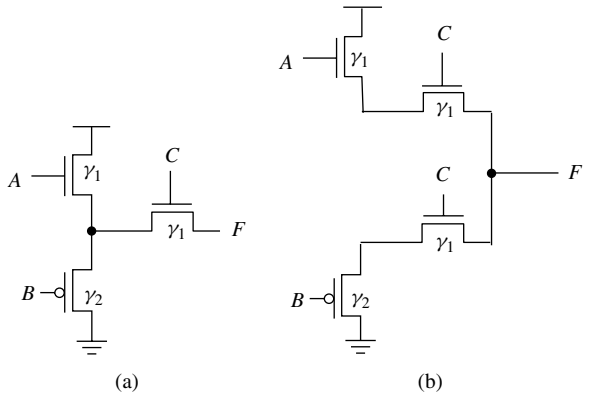


Figure 2.53 Comparing circuits.

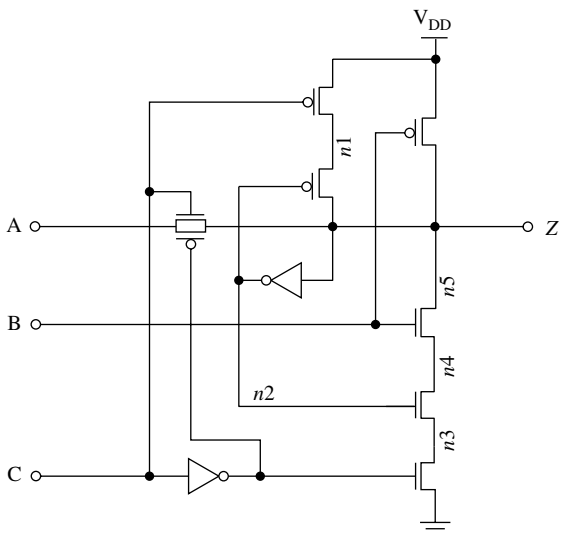


Figure 2.54 Determining the function.

to evaluate the following six input vectors: (0,0,1,1,0), (1,0,0,0,1), (1,1,0,1,0), (0,1,1,0,1), (1,1,1,0,1), (1,0,1,0,1). For the gate-level model, consider the number of event-driven evaluations if the circuit elements all have one unit of delay versus the number of evaluations if all elements have zero delay and the circuit is rank-ordered.

2.25 Create a ROBDD for the function $f = x_1 \cdot x_2 + x_3 \cdot x_4 + x_5 \cdot x_6$.

2.26 Create a ROBDD for the function $f = x_1 \cdot x_4 + x_2 \cdot x_5 + x_3 \cdot x_6$. Compare it with the ROBDD created in the previous problem. Can you generalize your conclusion?

2.27 Create ROBDDs for the equations f_1 and f_2, below. Use the Apply algorithm to compute $f_1 \oplus f_2$.

$$f_1 = x_1 \cdot x_2 \cdot x_3 + x_1 \cdot x_2 \cdot x_3 + x_1 \cdot x_2 \cdot x_3$$
$$f_2 = (x_1 \cdot x_2) \oplus x_3$$

2.28 Prove Shannon's expansion. *Hint*: Consider the function whose terms are expressed in standard sum-of-products form; that is, every variable appears in true or complement form in each term, and there is a term in the function corresponding to every row in the truth table that evaluates to 1.

REFERENCES

1. Druian, R. L., Functional Models for VLSI Design, *Proc. 20th D.A. Conf.*, 1983, pp. 506–514.
2. Falkoff, A. D., K. E. Iverson, and E. H. Sussenguth, Formal Description of System/360, *IBM Syst. J.*, 3, 1964, pp. 198–262.
3. Hill, F. J., and G. R. Peterson, *Computer Aided Logical Design: With Emphasis on VLSI*, 4th ed., John Wiley & Sons, New York, 1993.
4. Chu, Y., *Introduction to Computer Organization*, Prentice-Hall, Englewood Cliffs, NJ, 1970.
5. Duley, J. R., and D. L. Dietmeyer, A Digital System Design Language (DDL), *IEEE Trans. Comput.,* Vol. C-17, September 1968, pp. 850–861.
6. Kumar, Jainendra, Prototyping the M68060 for Concurrent Verification, *IEEE Des. Test*, Vol. 14, No. 1, January–March 1997, pp. 34–41.
7. Bryant, R. E., A Switch-level Model and Simulator for MOS Digital Systems, *IEEE Trans. Comput.*, Vol. C-33, No. 2, February 1984, pp. 160–177.
8. Sheffer, H. M., A Set of Five Independent Postulates for Boolean Algebras, *Trans. Am. Math. Soc.*, Vol. 14, 1913, pp. 481–488.
9. Huffman, D. A., The Synthesis of Sequential Circuits, *J. Franklin Inst.*, Vol. 257, 1954, pp. 161–190 and 275–303.
10. *The TTL Data Book*, 2nd ed., Texas Instruments, Dallas, TX, pp. 6–48.
11. Ulrich, E., and D. Hebert, Speed and Accuracy in Digital Network Simulation Based on Structural Modeling, *Proc. 19th D.A. Conf.*, 1982, pp. 587–593.
12. Eichelberger, E. B., Hazard Detection in Combinational and Sequential Switching Circuits, *IBM J. Res. Dev.*, Vol. 9, No. 2, March 1965, pp. 90–99.
13. Hardie, F. H., and R. J. Suhocki, Design and Use of Fault Simulation for Saturn Computer Design, *IEEE Trans. Electron. Comput.*, Vol. EC-16, No. 4, August 1967, pp. 412–429.
14. Thomas, Don, and Phil Moorby, *The Verilog Hardware Description Language*, 3rd ed., Kluwer, Boston, 1996.

15. Palnitkar, Samir, *Verilog HDL*, Prentice-Hall, Upper Saddle River, NJ, 1996.
16. IEEE 1364 Standard, *Verilog Hardware Description Language Reference Manual* (*LRM*), IEEE Standards Assoc., Piscataway, NJ.
17. Fantauzzi, G., An Algebraic Model for the Analysis of Logical Circuits, *IEEE Trans. Comput.*, Vol. C-23, No. 6, June 1974, pp. 576–581.
18. Phillips, N. D., and J. G. Tellier, Efficient Event Manipulation: The Key to Large Scale Simulation, *Proc. 1978 IEEE Int. Test Conf.*, pp. 266–273.
19. Ulrich, E. G., Exclusive Simulation of Activity in Digital Networks, *Commun. ACM*, Vol. 12, No. 2, February 1969, pp. 102–110.
20. Ulrich, E. G., Non-integral Event Timing for Digital Logic Simulation, *Proc. 14th D.A. Conf.*, 1976, pp. 61–67.
21. Bowden, K. R., Design Goals and Implementation Techniques for Time-Based Digital Simulation and Hazard Detection, *Proc. 1982 Int. Test Conf.*, pp. 147–152.
22. Hayes, J. P., A Logic Design Theory for VLSI, *Proc. Caltech Conf. VLSI*, January 1981, pp. 455–476.
23. Holt, D., and D. Hutchings, A MOS/LSI Oriented Logic Simulator, *Proc. 18th D.A. Conf.*, 1981, pp. 280–287.
24. Bryant, R. E., A Survey of Switch-Level Algorithms, *IEEE Des. Test*, August 1987, pp. 26–40.
25. Bryant, R. E., A Switch-Level Model of MOS Logic Circuits, *VLSI 81*, August 1981, pp. 329–340.
26. Bryant, R. E., A Switch-Level Model and Simulator for MOS Digital Systems, *IEEE Trans. Comput.*, Vol. C-33, No. 2, February 1984, pp. 160–177.
27. Bose, S., V. D. Agrawal, and T. G. Szymanski, Algorithms for Switch Level Delay Fault Simulation, *Proc. IEEE Int. Test Conf.*, 1997, pp. 982–991.
28. Akers, S. B., Binary Decision Diagrams, *IEEE Trans. Comput.*, Vol. C-27, No. 6, June 1978, pp. 509–516.
29. Lee, C. Y., Representation of Switching Circuits by Binary Decision Programs, *Bell Syst. Tech. J.*, Vol. 38, July 1959, pp. 985–999.
30. Aho, A. V., J. E. Hopcroft, and J. D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1974, pp. 51–55.
31. Bryant, E. Randal, Graph-Based Algorithms for Boolean Function Manipulation, IEEE Trans. Comput., August 1986, Vol. C-35, No. 8, pp. 677–691.
32. Miczo, A. et al., The Effects of Modeling on Simulator Performance, *IEEE Des. Test*, Vol. 4, No. 2, April 1987, pp. 46–54.
33. Hitchcock, R. B., Timing Verification and the Timing Analysis Program, *Proc. 19th D.A. Conf.*, 1982, pp. 594–604.
34. Wold, M. A., Design Verification and Performance Analysis, *Proc. 15th D.A. Conf.*, 1978, pp. 264–270.
35. Ng, P. et al., A Timing Verification System Based on Extracted MOS/VLSI Circuit Parameters, *Proc. 18th D.A. Conf.*, 1981, pp. 288–292.

CHAPTER 3

Fault Simulation

3.1 INTRODUCTION

Thus far simulation has been considered within the context of design verification. The purpose was to determine whether or not the design was correct. Were all the key control signals of the design checked out? What about the data paths, were all the "corners" or endpoints checked out? Are we confident that all likely combinations of events have been simulated and that the circuit model responded correctly? Is the design ready to be taped out?

We now turn our attention to simulation as it relates to manufacturing test. Here the objective is to create a test program that uncovers defects and performance problems that occur during the manufacturing process. In addition to being thorough, a test program must also be efficient. If design verification involves a large number of redundant simulations, there is unnecessary delay in moving the design to tape-out. If the manufacturing test program involves creation of redundant test stimuli, there is delay in migrating the test program to the tester. However, stimuli that do not improve test thoroughness also add recurring costs at the tester because there is the cost of providing storage for all those test stimuli as well as the cost of applying the excess stimuli to every chip that is manufactured.

There are many similarities between design verification and manufacturing test program development, despite differences in their objectives. In fact, design verification test suites are often used as part (or all) of the manufacturing test program. In either case, the first step is to create a circuit model. Then, input stimuli are created and applied to the model. For design verification, the response is examined to ascertain that it matches the expected response. For test program development the response is examined to ensure that faults are being detected. This process, "apply stimuli–monitor response," is continued until, based on some criteria, the process is determined to be complete.

Major differences exist between manufacturing test program development and design verification. Test programs are often constrained by physical resources, such as the tester architecture, the amount of tester memory available, or the amount of

Digital Logic Testing and Simulation, *Second Edition*, by Alexander Miczo
ISBN 0-471-43995-9 Copyright © 2003 John Wiley & Sons, Inc.

tester time available to test each individual integrated circuit (IC). The manufacturing test usually can only observe activity at the I/O pins and is considerably less flexible in its ability to create input vectors because of limitations on timing generators and waveform electronics in the tester. Design verification, using a hardware design language (HDL) and conducted within a testbench environment, has virtually infinite flexibility in its ability to control details such as signal timings and relationships between signals. Commands exist to monitor and display the contents of registers and internal signals during simulation. Messages can be written to the console if illegal events (e.g., setup or hold violations) occur inside the model.

Another advantage that design verification has over manufacturing test is the fact that signal paths from primary inputs to primary outputs can be verified piecemeal. This simply means that a logic designer may check out a path from a particular internal register to an output port during one part of a test and, if satisfied that it works as intended, never bother to exercise that path again. Later, with other objectives in mind, the designer may check out several paths from various input ports to the aforementioned register. This is perfectly acceptable as a means of determining whether or not signal paths being checked out are designed correctly. By contrast, during a manufacturing test the values that propagate from primary inputs to internal registers must continue to propagate until they reach an output port where they can be observed by the tester. Signals that abruptly cease to propagate in the middle of an IC or PCB reveal nothing about the physical integrity of the device.

An advantage that manufacturing test has over design verification is the assumption, during manufacturing test development, that the design is correct. The assumption of correctness applies not only to logic response, but also to such things as setup and hold times of the flip-flops. Hence, if some test stimuli are determined by the fault simulator to be effective at detecting physical defects, they can be immediately added to the production test suite, and there is no need to verify their correctness. By way of contrast, during design verification, response to all stimuli must be carefully examined and verified by the logic designer.

Some test generation processes can be automated, for example, combinational blocks such as ALUs can be simulated using large suites of random stimuli. Simulation response vectors can be converted from binary to decimal and compared to answers that were previously calculated by other means. For highly complex control logic, the process is not so simple. Given a first-time design, where there is no existing, well-defined behavior that can be used as a "gold standard," all simulation response files must be carefully inspected. In addition to correct logic response, it will usually be necessary to verify that the design performs within required time constraints.

3.2 APPROACHES TO TESTING

Testing digital logic consists of applying stimuli to a device-under-test (DUT) and evaluating the response to determine whether the device is responding correctly. An important part of the test is the creation of effective stimuli. The stimuli can be created in one of three ways:

1. Generate all possible combinations.
2. Develop test programs that exercise the functionality of the design.
3. Create test sequences targeted at specific faults.

Early approaches to creation of stimuli, circa 1950s, involved the application of all possible binary combinations to device inputs to perform a complete functional verification of the device. Application of 2^n test vectors to a device with n inputs was effective if n was small and if there were no sequential circuits on the board. Because the number of tests, 2^n, grows exponentially with n, the number of tests required increases rapidly, so this approach quickly ran out of steam.

In order to exercise the functionality of a device, such as the circuit in Figure 3.1, a logic designer or a test engineer writes sequences of input stimuli intended to drive the device through many different internal states, while varying the conditions on the data-flow inputs. Data transformation devices such as the ALU perform arithmetic and logic operations on arguments provided by the engineer and these, along with other sequences, can be used to exercise storage devices such as registers and flip-flops and data routing devices such as multiplexers. If the circuit responds with all the correct answers, it is tempting to conclude that the circuit is free of defects. That, however, is the wrong conclusion because the circuit may have one or more defects that simply were not detected by the applied stimuli. This lack of accountability is a major problem with the approach—there is no practical way to evaluate the effectiveness of the test stimuli. Effectiveness can be estimated by observing the number of products returned by the customer, so-called "tester escapes," but that is a costly solution. Furthermore, that does not solve the problem of diagnosing the cause of the malfunction.

In 1959, R. D. Eldred[1] advocated testing hardware rather than function. This was to be done by creating tests for specific faults. The most commonly occurring faults would be modeled and input stimuli created to test for the presence or absence of each of these faults. The advantages of this approach are as follows:

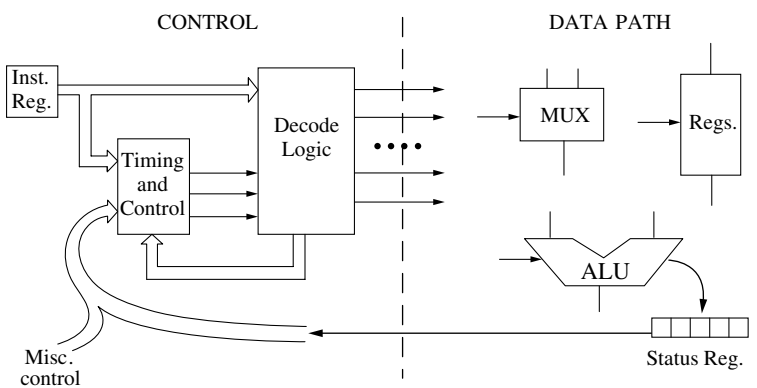


Figure 3.1 Functional view of CPU.

1. Specific tests can be created for faults most likely to occur.
2. The effectiveness of a test program can be measured by determining how many of the commonly occurring faults are detected by the set of test vectors created.
3. Specific defects can be associated with specific test vectors. Then, if a DUT responds incorrectly to a test vector, there is information pointing to a faulty component or set of components.

This method advocated by Eldred has become a standard approach to developing tests for digital logic failures.

3.3 ANALYSIS OF A FAULTED CIRCUIT

A prerequisite for being able to test for faults in a digital circuit is an understanding of the kinds of faults that can occur and the consequences of those faults. To that end, we will analyze the circuit of Figure 3.2. We hypothesize the presence of a fault in the circuit, namely, a short across resistor R_4. Then a test will be created that is capable of detecting the presence of that fault.

3.3.1 Analysis at the Component Level

In the analysis that follows, the positive logic convention will be used. Any voltage between ground (Gnd) and +0.8 V represents a logic 0. A voltage between +2.4 V and +5.0 V (Vcc) represents a logic 1. A voltage between +0.8 V and +2.4 V represents an indeterminate state, indicated by the symbol X. The bipolar NPN transistors Q_1 through Q_6 behave like on/off switches when used in digital circuits. A low voltage on the base cuts off a transistor so that it cannot conduct. The circuit behaves as though there were an open circuit between the emitter and collector. A high voltage on the base causes the transistor to conduct, and the circuit behaves as though a direct connection exists between the emitter and collector.

With these definitions, it is possible to analyze the fault and its effects on the circuit. Note that with the resistor shorted, the base of Q_3 is held at ground. It will not conduct and behaves like an open switch. This causes the voltage at the collector of Q_3 to remain high, a logic 1, which in turn causes the base of Q_5 and the emitter of Q_4 to remain high. Q_4 will not be able to conduct because its base cannot be made more positive than its emitter. However, Q_5 is capable of conducting, depending on the voltage applied to its emitter by Q_6.

If Z is high ($Z = 1$), the positive voltage on the base of Q_6 causes it to conduct; hence it is in effect shorted to ground. Therefore, the base of Q_5 is more positive than the emitter, transistor Q_5 conducts, and the output goes low. If Z is low ($Z = 0$), Q_6 is cut off. Since it does not conduct, the base and emitter of Q_5 are at the same potential, and it is cut off. Therefore the output of Q_5 goes high and the output of F is at logic 1. As a result of the fault, the value at output F is the complement of the value at input Z and is totally independent of any signals appearing at X_1, X_2, Y_1, and Y_2.

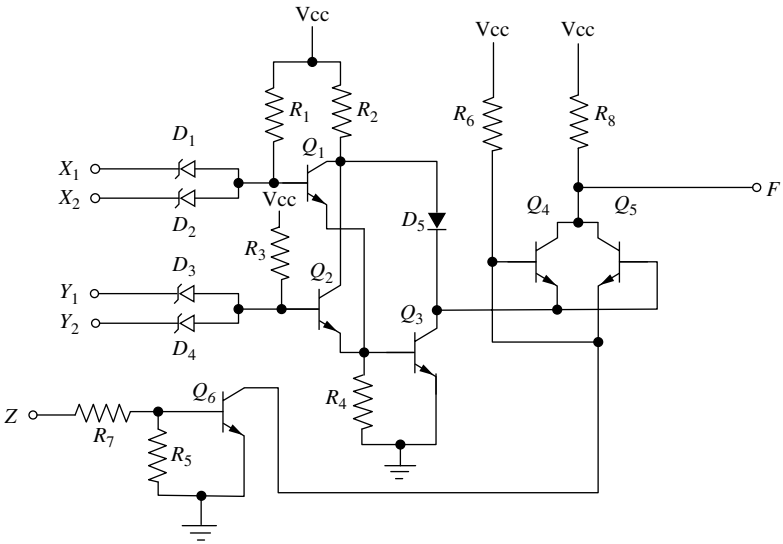


Figure 3.2 Component-level circuit.

We now know how the circuit behaves when the fault is present. But how do we devise input stimuli that will tell us if the fault is present? It is assumed that the output F is the only point in the circuit that can be observed, internal nodes cannot be probed. This restriction tells us that the only way to detect the fault is to create input stimuli for which the output response is a function of the presence or absence of the fault. The response of the circuit with the fault will then be opposite that of the fault-free circuit.

First, consider what happens if the fault is not present. In that case, the output is dependent not only on Z, but also on X_1, X_2, Y_1, and Y_2. If the values on these inputs cause the output of Q_3 to go high, the faulted circuit cannot be distinguished from the fault-free circuit, because the circuits produce identical signals at the output of Q_3 and hence identical signals at the output F. However, if the output of Q_3 is low, then an analysis of the circuit as done previously reveals that the output F equals Z. Therefore, when Q_3 is low, the signal at F is opposite what it would be if the fault were present, so we conclude that we want to apply a signal to the base of Q_3 that causes the collector to go low. A positive signal on the base will produce the desired result. Now, how do we get a high signal on the base of Q_3? To determine that, it is necessary to analyze the circuits preceding Q_3.

Consider the circuit made up of Q_1, R_1, D_1, and D_2. If either X_1 or X_2 is at logic 0, then the base of Q_1 is at ground potential; hence Q_1 acts like an open switch. Likewise, if Y_1 or Y_2 is at logic 0, then Q_2 acts like an open switch. If both Q_1 and Q_2 are open, then the base of Q_3 is at ground. But we wanted a high signal on the base of Q_3. If either Q_1 or Q_2 conducts, then there is a complete path from ground through R_4, through Q_1 or Q_2, through R_2 to Vcc. Then, with the proper resistance values on R_1, R_2, and R_4, a high-voltage signal appears at the base of Q_3. Therefore, we conclude

that there must be a high signal on X_1 and X_2 or Y_1 and Y_2 (or both) in order to determine whether or not the fault is present. Note that we must also know what signal is present on input Z. With $X_1 = X_2 = 1$ or $Y_1 = Y_2 = 1$, the output F assumes the same value as Z if the fault is not present and assumes the opposite value if the fault is present.

3.3.2 Gate-Level Symbols

Analyzing circuits at the transistor level in order to calculate signal values that distinguish between good and faulty circuits is quite tedious. It requires circuit engineers capable of analyzing complex circuits because, within a given technology, there are many ways to design circuits at the component level to accomplish the same end result, from a logic standpoint. In a large circuit with thousands of individual components, it is not obvious, exactly what logic function is being performed by a particular group of components. Further complicating the task is the fact that a circuit might be implemented in one of several technologies, each of which has its own unique way to perform digital logic operations. For instance, in Figure 3.2 the subcircuit made up of D_1 through D_5, Q_1 through Q_3, and R_1 through R_3 constitutes an AND-OR-Invert circuit. The same subcircuit is represented in a complementary metal–oxide semiconductor (CMOS) technology by the circuit in Figure 3.3. The two circuits perform the same logic operation but bear no physical resemblance to one another!

3.3.3 Analysis at the Gate Level

The complete gate equivalent circuit to the circuit in Figure 3.2 is shown in Figure 3.4. We already stated that Q_1 through Q_5, D_1 through D_5, and R_1 through R_3 constitute an AND-OR-Invert. The components Q_3, R_5, and R_6 constitute an Inverter and the transistors Q_4, Q_5 together make up an Exclusive-NOR (EXNOR, an exclusive-OR with its output complemented.) Hence, the circuit of Figure 3.2 can be represented by the logic diagram of Figure 3.4.

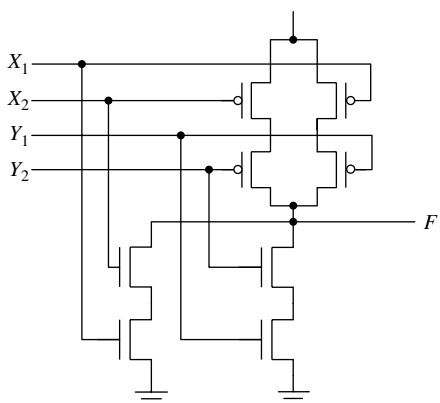


Figure 3.3 CMOS AND-OR-Invert.

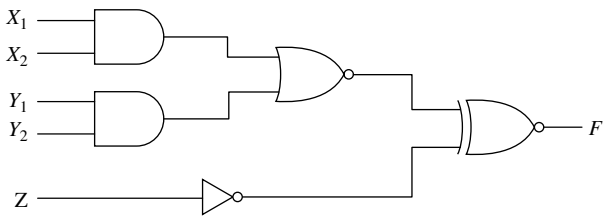


Figure 3.4 The gate equivalent circuit.

Now reconsider the fault that we examined previously. When R_4 was shorted, the output of Q_3 could not be driven to a low state. That is equivalent to the NOR gate output in the circuit of Figure 3.4 being stuck at a logic 1. Consequently, we want to assign inputs that will cause the output of the NOR gate, when fault-free, to be driven low. This requires a 1 on one of the two inputs to the gate. If the upper input is arbitrarily selected and required to generate a logic 1, then the upper AND gate must generate a logic 1, requiring that inputs X_1 and X_2 must both be at logic 1. As before, a known value must be assigned to input Z so that we know what value to expect at primary output F for the fault-free and the faulted circuits. The reader will (hopefully) agree that the circuit representation of Figure 3.4 is much easier to analyze.

The circuit representation of Figure 3.4, in addition to being easier to work with and requiring fewer details to keep track of, has the additional advantage of being understandable by people who are familiar with logic but not familiar with transistor-level behavior. Furthermore, it is universal; that is, a circuit can be represented in terms of these symbols regardless of whether the circuit is implemented in MOS, TTL, ECL, or some other technology. As long as the circuit can be logically modeled, it can be represented by these symbols. Another important advantage of this representation, as will be seen, is that computer algorithms can be defined on these logic operations which are, for the most part, independent of the particular technology chosen to implement the circuit. If the circuit can be expressed in terms of these symbols, then the circuit description can be processed by the computer algorithms.

3.4 THE STUCK-AT FAULT MODEL

A circuit composed of resistors, diodes, and transistors can be represented as an interconnection of logic gates. If this gate-level model is altered so as to represent a faulted circuit, then the behavior of the faulted circuit can be analyzed and tests developed to distinguish it from the fault-free circuit. But, for what kind of faults should tests be created? The wrong answer can result in an extremely difficult problem. As a minimum, a fault model must possess the following four properties:

1. It must correspond to real faults.
2. It must have adequate granularity.
3. It must be accountable.
4. It must be easily automated.

The fault in the circuit of Figure 3.2 was represented as a NOR gate output stuck-at-1 (SA1). What happens if diode D_1 is open? If that fault is present, it is not possible to pull the base of Q_1 to ground potential from input X_1. Therefore input 1 of the AND gate, represented by D_1, D_2, R_1 and Q_1, is SA1. What happens if there is an open from the common connection of the emitters of Q_1 and Q_2 to the emitter of Q_1? Then, there is no way that Q_1 can provide a path from ground, through R_4, Q_1, and R_2 to Vcc. The base of Q_3 is unaffected by any changes in the AND gate. Since the common connection of Q_1 and Q_2 represents an OR operation (called a wired-OR or DOT-OR), the fault is equivalent to an OR gate input stuck-at-0 (SA0).

The stuck-at fault model corresponds to real faults, although it clearly does not represent all possible faults. It has been well known for many years that test programs based on the stuck-at model can detect all stuck-at faults and still fail to identify all defective parts.[2] The term *granularity* refers to the resolution or level of detail at which a model represents faults. A model should represent most of the faults that occur within gate-level models. Then, if a test detects all of the modeled faults, there is a high probability that it will detect all of the actual physical defects that may occur. A fault model with fine granularity is more useful than a model with coarse granularity, since a test may detect all faults from a fault class with coarse granularity and still miss many microscopic defects.

An n-input combinational circuit can implement any of 2^{2^n} functions. To verify with *absolute* certainty that the circuit implements the correct function, it is necessary to apply all 2^n input combinations and confirm that the circuit responds correctly to each stimulus. That could take an enormous amount of time. If a randomly chosen subset of all possible combinations is applied, there is no way of measuring the effectiveness of the test, unless a correlation can be shown between the number of test pattern combinations applied and the effectiveness of the test. By employing a fault model, we can account for the faults, determining via simulation which faults were detected and on what vector they were first detected.

Given that we want to use fault models, as well as employ simulation to determine how many faults are detected by a given test program, what fault model should be chosen? We could assign a status for each of the nets in a circuit, according to the following list:

fault-free
stuck-at-1
stuck-at-0

Given a circuit containing m nets that interconnect the various components, if all possible combinations are considered, then there are 3^m circuits described by the m nets and the three possible states of each net. Of these possibilities, only one corresponds to a completely fault-free circuit.

If all possible combinations of shorts between nets are considered, then there are

$$\sum_{i=2}^{m} \binom{m}{i} = 2^m - m - 1$$

shorts that could occur in an actual circuit. The reader will note that we keep bumping into the problem of "combinatorial explosion"; that is, the number of choices or problems to be solved explodes. To attempt to test for every stuck-at or short fault combination is clearly impractical.

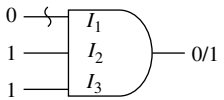
As it turns out, many component defects can be represented as stuck-at faults on inputs or outputs of logic gates. The SAx, $x \in \{0,1\}$, fault model has become universal. It has the attraction that it has sufficient granularity that a test which detects a high percentage of the stuck-at faults will detect a high percentage of the real defects that occur. Furthermore, the stuck-at model permits enumeration of faults. For an n-input logic gate, it is possible to identify a specific set of faults, as well as their effect on circuit behavior. This permits implementation of computer algorithms targeted at those faults. Furthermore, by knowing the exact number of faults in a circuit, it is possible to keep track of those that are detected by a test, as well as those not detected. From this information it is possible to create an effectiveness measure or figure of merit for the test.

The impracticality of trying to test for every conceivable combination of faults in a circuit has led to adoption of the *single-fault assumption*. When attempting to create a test, it is assumed that a single fault exists. Most frequently, it is assumed that an input or output of a gate is SA1 or SA0. Many years of experience with the stuck-at fault model by many digital electronics companies has demonstrated that it is effective. A good stuck-at test which detects all or nearly all single stuck-at faults in a circuit will also detect all or nearly all multiple stuck-at faults and short faults. There are technology-dependent faults for which the stuck-at fault model must be modified or augmented; these will be discussed in a later chapter.

Another important assumption made in the industry is the reliance on solid failures; intermittent faults whose presence depends on environmental or other external factors such as temperature, humidity, or line voltage are assumed to be solid failures when creating tests. In the following paragraphs, fault models are described for AND, OR, Inverter, and the tri-state buffer. Fault models for other basic circuits can be deduced from these. Note that these gates are, in reality, low-level behavioral models that might be implemented in CMOS, TTL, ECL, or any other technology. The gate-level function hides the transistor level implementation details, so the tests described here can be viewed as behavioral test programs; that is, all possible combinations on the inputs and outputs of the gates are considered, and those that are redundant or otherwise add no value are deleted.

3.4.1 The AND Gate Fault Model

The AND gate is fault-modeled for inputs SA1 and the output SA1 and SA0. This results in $n + 2$ tests for an n-input AND gate. The test for an input SA1 consists of putting a logic 0 on the input being tested and logic 1s on all other inputs (see Figure 3.5). The input being tested is the controlling input; it determines what value appears on the output. If the circuit is fault-free, the output goes to a logic 0; and if the fault is present, the output goes to a logic 1. Note that if any of the inputs, other than the one being tested, has a 0 value, that 0 is called a *blocking value*, since it prevents the test for the faulted pin from propagating to the output of the gate.



	I_1	I_2	I_3	G	F_1	F_2	F_3	F_4	F_5
$F_1 - I_1$ SA1	0	0	0	0	0	0	0	0	1
$F_2 - I_2$ SA1	0	0	1	0	0	0	0	0	1
$F_3 - I_3$ SA1	0	1	0	0	0	0	0	0	1
$F_4 - Out$ SA0	0	1	1	0	1	0	0	0	1
$F_5 - Out$ SA1	1	0	0	0	0	0	0	0	1
	1	0	1	0	0	1	0	0	1
	1	1	0	0	0	0	1	0	1
	1	1	1	1	1	1	1	0	1

Figure 3.5 AND gate with stuck-at faults.

An input pattern of all 1s will test for the output SA0. It is not necessary to explicitly test for an output SA1 fault since any input SA1 test will also detect the output SA1. However, an output SA1 can be detected without detecting any input SA1 fault if two or more inputs have logic 0s on their inputs, therefore it can be useful to retain the output SA1 as a separate fault. When tabulating faults detected by a test, counting the output as tested when none of the inputs is tested provides a more accurate estimate of fault coverage. Note that a SA0 fault on any input will produce a response identical to that of fault F_4. The all-1s test for fault F_4 will detect a SA0 on any input; hence, it is not necessary to test explicitly for a SA0 fault on any of the inputs.

3.4.2 The OR Gate Fault Model

An n-input OR gate, like the AND gate, requires $n + 2$ tests. However, the input values are the complement of what the values would be for an AND gate. The input being tested is set to 1 and all other inputs are set to 0. The test is checking for the input SA0. The all-0s input tests for the output SA1 and any input SA1. A logic 1 on any input other than the input being tested is a blocking value for the OR gate.

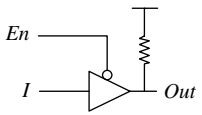
3.4.3 The Inverter Fault Model

The Inverter can be modeled with a SA0 and SA1 on its output, or it could be modeled with SA1 and SA0 on its input. If it fails to invert, perhaps caused by a short across a transistor, and if both stuck-at faults are detected, the short fault will be detected by one of the stuck-at tests.

3.4.4 The Tri-State Fault Model

The Verilog hardware description language recognizes four tri-state gates: bufif0, bufif1, notif0, and notif1. The bufif0 (bufif1) is a buffer with an active low (high) control input. The notif0 (notif1) is an inverter with an active low (high) control input. Figure 3.6 depicts the bufif0. Behavior of the others can be deduced from that of the bufif0.

Five faults are listed in Figure 3.6, along with the truth table for the good circuit G, and the five faults F_1 through F_5. Stuck-at faults on the input or output, F_3, F_4, or F_5, can be detected while the enable input, En, is active. Stuck-at faults on the enable input present a more difficult challenge.



F_1 – *En* SA0
F_2 – *En* SA1
F_3 – *I* SA0
F_4 – *I* SA1
F_5 – *Out* SA1

En	*I*	*G*	F_1	F_2	F_3	F_4	F_5
0	0	0	0	Z	0	1	1
0	1	1	1	Z	0	1	1
1	0	Z	0	Z	Z	Z	1
1	1	Z	1	Z	Z	Z	1

Figure 3.6 bufif0 with faults.

If fault F_1 occurs, the enable is always active, so the bufif0 is always driving the bus to a logic 1 or 0. There are two possibilities to consider: One possibility is that no other device is actively driving the bus. To detect a fault, it is necessary to have the fault-free and faulty circuits produce different values at the output of the bufif0. But, from the truth table it can be seen that the only way that good circuit *G* and faulty circuit F_1 can produce different values is if *G* produces a Z on the output and F_1 produces a 1 or 0. This can be handled by connecting a pullup or pulldown resistor to the bus. Then, in the absence of a driving signal, the bus floats to a weak 1 or 0. With a pullup resistor—that is, a resistor connected from the bus to V_{DD} (logic 1)—a logic 0 on the input of the bufif0 forces the output to a value opposite that caused by the pullup.

The other possibility is that another bus driver is simultaneously active. Eventually, the two drivers are going to drive the bus to opposite values, causing *bus contention*. During simulation, contention causes the bus to be assigned an indeterminate X. If the signal makes it to an output, the X can only be a *probable detect*. In practice, the contending values represent a short, or direct connection, between ground and power, and the excess current causes the IC to fail completely.

The occurrence of fault F_2 causes the output of the bufif0 to always be disconnected from the bus. When the enable on the good circuit *G* is set to 0, the fault-free circuit can drive a 1 or 0 onto the bus, whereas the faulty circuit is disconnected; that is, it sees a Z on the bus. This propagates through other logic as an X, so if the X reaches an output, the fault F_2 can only be recorded as a probable detect. As in the previous paragraph, a pullup or pulldown can be used to facilitate a *hard detect*—that is, one where the good circuit and faulty circuit have different logic values.

3.4.5 Fault Equivalence and Dominance

When building fault lists, it is often the case that some faults are indistinguishable from others. Suppose the circuit in Figure 3.7 is modeled with an SA0 fault on the output of gate *B* and all eight input combinations are simulated. Then that fault is removed and the circuit is modeled with an SA0 fault on the top input of gate *D* and resimulated. It will be seen that the circuit responds identically at output *Z* for both of the faults. This is not surprising since the output of *B* and the input of *D* are tied to the same net. We say that they are *equivalent faults*. Two faults are equivalent if there is no logic test that can distinguish between them. More precisely, if T_a is the set of

tests that detect fault a and T_b is the set of tests that detect fault b, and if $T_a = T_b$, then it is not possible to distinguish a from b. A set of faults that are equivalent form an equivalence class. In such instances, a single fault is selected to represent the equivalence class of faults.

Although a tester cannot logically distinguish which of several equivalent faults causes an error response at an output pin, the fact that some equivalence classes may contain several stuck-at faults, and others may contain a single fault, is sometimes used in industry to bias the fault coverage. If an equivalence class representing five stuck-at faults is undetected, it is deemed, in such cases, to have as much effect on the final fault coverage as five undetected faults from equivalence classes containing a single fault. From a manufacturing standpoint, this weighting of faults reflects the fact that not all faults are equal; a fault class with five stuck-at faults has a higher probability of occurring than a fault class with a single stuck-at fault.

In a previous subsection it was pointed out that the fault list for an n-input AND gate consisted of $n + 2$ entries. However, any test for an input i SA1 simultaneously tested the output for a SA1. The converse does not hold; a test for a SA1 on the output need not detect any of the input SA1 faults. We say that the output SA1 fault dominates the input SA1 fault. In general, fault a dominates fault b if $T_b \subseteq T_a$. From this definition it follows that if fault a dominates fault b, then any test that detects fault b will detect fault a.

A function F is *unate* in variable x_i if the variable x_i appears in the sum-of-products expression for F in its true or complement form but not both. The concept of *fault dominance* for logic elements can now be characterized:[3]

Theorem 3.1 Given a combinational circuit $F(x_1, x_2, ..., x_n)$, a dominance relation exists between faults on the output and input x_i iff F is unate in x_i.

A function is *partially symmetric* in variables x_i and x_j if $F(x_i, x_j) = F(x_j, x_i)$. A function is *symmetric* if it is partially symmetric for all input variable pairs x_i, x_j. With those definitions we have:

Theorem 3.2 If a logic gate is partially symmetric for inputs i and j, then either faults on those inputs are equivalent or no dominance relation holds.

Theorem 3.3 In a fan-out free circuit realized by symmetric, unate gates, tests designed to detect stuck-at faults on primary inputs will detect all stuck-at faults in the circuit.

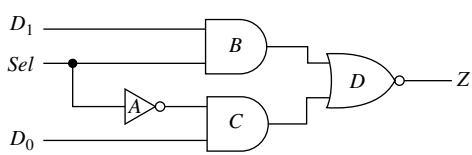


Figure 3.7 Equivalent and dominant faults.

Equivalence and dominance relations are used to reduce fault list size. Since computer run time is affected by fault list size, the reduction of the fault list, a process called *fault collapsing*, can reduce test generation and fault simulation time. Consider the multiplexer of Figure 3.7. An SA0 fault on the output of NOR gate *D* is equivalent to an SA1 fault on any of its inputs, and an SA1 fault on the output of *D* dominates an SA0 fault on any of its inputs. SA0 faults on the inputs to gate *D*, in turn, are equivalent to SA0 faults on the outputs of gates *B* and *C*. Therefore, for the purposes of detection, if SA0 faults on the inputs of gate *D* are detected, SA0 faults on the outputs of gates *B* and *C* can be ignored.

3.5 THE FAULT SIMULATOR: AN OVERVIEW

The use of fault simulation is motivated by a desire to minimize the amount of defective product shipped to customers. Recall, from Chapter 1, that defect level is a function of process yield and the thoroughness of the test applied to the ICs. It is obvious that the amount of defective product (tester escapes) can be reduced by improving yield or by improving the test. To improve a test, it is first necessary to quantify its effectiveness. But, how?

Fault simulation is the process of measuring the quality of a test. Test stimuli that will eventually be applied to the product on a tester are themselves first evaluated by applying them to circuit models that have been slightly altered to imitate the effects of faults. If the response at the circuit outputs, as determined by simulation, differs from the response of the circuit model without the fault, then the fault is detectable by those stimuli. After the process is performed for a sufficient number of modeled faults, an estimate *T*, called the *fault coverage*, or *test coverage*, is computed. The equation is

$$T = (\#\text{ faults detected})/(\#\text{ faults simulated})$$

The variable *T* reflects the quality or effectiveness of the test stimuli. Fault simulation is performed on a *structural* model, meaning that the model describes the system in terms of realizable physical components. The term can, however, refer to any level except behavioral, depending upon whether the designer was creating a circuit using geometrical shapes or functional building blocks. The fault simulator is a structural level simulator in which some part of the structural model has been altered to represent behavior of a fault. The fault simulator is instrumented to keep track of all differences in response between the unfaulted and the faulted circuit.

Fault simulation is most often performed using gate-level models, because of their granularity, although fault simulation can also be performed using functional or circuit level models. The stuck-at fault model, in conjunction with logic gates, makes it quite easy to automatically inject faults into the circuit model by means of a computer program. Fault simulation serves several purposes besides evaluating stimuli:

- It confirms detection of a fault for which an ATPG generates a test.
- It computes fault coverage for specific test vectors.

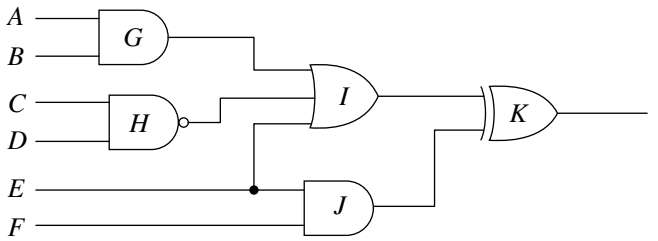


Figure 3.8 Circuit with fault.

- It provides diagnostic capability.
- It identifies areas of a circuit where fault coverage is inadequate.

Confirm Detection When creating a test, an automatic test pattern generator (ATPG) makes simplifying assumptions. By restricting its attention to logic behavior and ignoring element delay times, the ATPG runs the risk of creating test vectors that are susceptible to races and hazards. A simulator, taking into account element delays and using hazard and race detection techniques, may detect anomolous behavior caused by the pattern and conclude that the fault cannot be detected with certainty.

Compute Fault Coverage The ability to identify all faults detected by each vector can reduce the number of iterations through an ATPG. As will be seen in the next chapter, an ATPG targets specific faults. If a fault simulator identifies faults that were detected incidentally by a vector created to detect a particular fault, there is no need to create test vectors to detect those other faults. In addition, the fault simulator can identify vectors that detect no faults, potentially reducing the size of a test program.

Example Suppose the pattern $A,B,C,D,E,F = (0,1,1,1,0,0)$ is created to test for the output of gate H SA1 in the circuit of Figure 3.8. Simulating the fault-free circuit produces an output of 0. Simulating the same circuit with a SA1 on the output of H produces a 1 on the circuit output; hence the fault is detected. But, when the effects of a SA1 on the upper input to gate G are simulated using the same pattern, we find that this fault also causes the circuit to respond with a 1 and therefore is detected by the pattern. Several other faults are detected by the pattern. We leave it as an exercise for the reader to find them. ■■

Diagnose Faults Fault diagnosis was more relevant in the past when many discrete parts were used to populate PCBs. When repairing a PCB, there was an economic incentive to obtain the smallest possible list of suspect parts. Diagnosis can also be useful in narrowing down the list of suspect logic elements when debugging first silicon during IC design. When a dozen masks or more are used to create an IC with hundreds of thousands of switching elements, and the mask set has a flaw that causes die to be manufactured incorrectly, knowing which vector(s) failed and knowing which faults are detected by those vectors can sometimes significantly reduce the scope of the search for the cause of the problem.

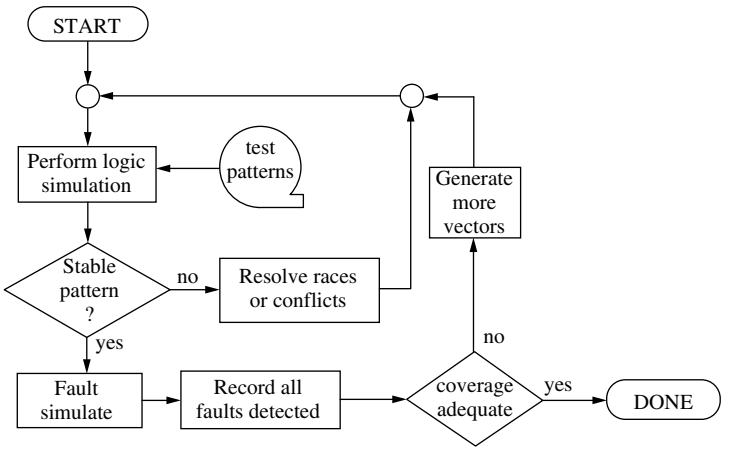


Figure 3.9 Test stimuli evaluation.

Consider again the circuit in Figure 3.8. If the circuit correctly responds with a 0 to the previous input pattern, there would not have been a SA1 fault on the output of gate *H*. If the next pattern applied is *A,B,C,D,E,F* = (0,0,1,1,0,1) and an incorrect response occurs, the stuck-at-1 on the output of gate *H* would not be suspect. By eliminating the signal path that contains gate *H* as a candidate, the amount of work involved in identifying the cause of the defect has been reduced.

Identify Areas of Untesteds When a test engineer writes stimuli for a circuit, he may expend much effort in one area of the circuit but very little effort in another area. The fault simulator can provide a list of faults not yet detected by test stimuli and thus encourage the engineer to work in an area of the circuit where very few faults have been detected. Writing test vectors targeted at faults in those areas frequently gives a quick boost to the fault coverage.

The overall test program development workflow, in conjunction with a fault simulator, is illustrated in Figure 3.9. The test vectors may be created by an ATPG or supplied by the logic designer or a diagnostic engineer. The ATPG is fault-oriented, it selects a fault from a list of fault candidates and attempts to create a test for the fault. Because stimuli created by the ATPG are susceptible to races and hazards, a logic simulation may precede fault simulation in order to screen the test stimuli. If application of the stimuli causes many races and hazards, it may be desirable to repair the stimuli before proceeding with fault simulation.

After each test vector has been fault-simulated, faults which cause an output response that differs from the correct response are checked off in the fault list, and their response at primary outputs may be recorded in a data base for diagnostic purposes. The circuits used here for illustrative purposes usually have a single output, but real circuits have many outputs and several faults may be detected in a given pattern, with each fault possibly producing a different response at the primary outputs.

By recording the output response to each fault, diagnostic capability can be significantly enhanced. After recording the results, if fault coverage is not adequate, the process is continued. Additional vectors are generated; they are checked for races and conflicts and then handed off to the fault simulator.

3.6 PARALLEL FAULT PROCESSING

Section 2.6 contains a listing for a compiled simulator that uses the native instruction set of the 80 × 86 microprocessor to simulate the circuit of Figure 2.9. With just some slight modifications, that same simulator can be instrumented to perform fault simulation. In fact, as we shall see, a fault simulator can be viewed conceptually as a logic simulator augmented with some additional capabilities, namely, the ability to keep track of differences in response between two nearly identical circuits.

For purposes of contrast, we discuss briefly the *serial* fault simulator; it is the simplest form of fault simulation. In this method a single fault is injected into the circuit model and simulated with the same stimuli that were applied to the fault-free model. The response at the outputs is compared to the response from the fault-free circuit. If the fault causes an output response that differs from the expected response, the fault is marked as detected by the applied stimuli. After the fault has been detected, or after all stimuli have been simulated, the fault is removed and another fault is injected into the circuit model. Simulation is again performed. This is done for all faults of interest, and then the fault coverage T is computed.

In the serial fault simulator, fault injection can be achieved for a logic gate simply by deleting an input. An entry in the descriptor cell of Figure 2.21 is blanked out and the input count is decremented. When a net connected to the input of an AND gate is deleted from the list of inputs to that AND gate, the logic value on that net no longer has an effect on the AND gate; hence the AND gate behaves as though that input were stuck-at-1. Likewise, deleting an input to the OR gate causes that input to behave as though it were stuck-at-0.

3.6.1 Parallel Fault Simulation

When the 80 × 86 compiled simulator described in Section 2.6 processed a circuit, it manipulated bytes of data. For ternary simulation, one bit from each of two bytes can be used to represent a logic value. This leaves seven bits unused in each byte. The *parallel* fault simulator can take advantage of the unused bits to simulate faulted circuits in parallel with the good circuit. It does this by letting each bit in the byte represent a different circuit. The leftmost bit (bit 7) represents the fault-free circuit. The other seven bits represent circuits corresponding to seven faults in the fault list. In order to use these extra bits, they must be made to represent values that exist in faulted circuits. This is accomplished by "bugging the simulator." Fault injection in the simulator must be accomplished in such a way that individual faults affect only a single bit position.

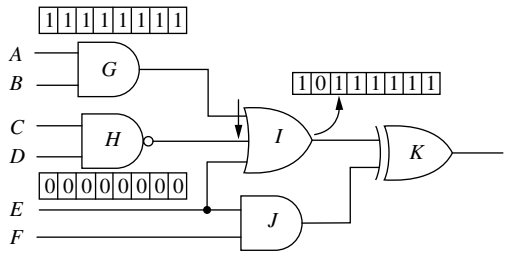


Figure 3.10 Parallel fault simulation.

Example OR gate *I* in Figure 3.10 is modeled with a SA0 on its top input. Bit 7 represents the fault-free circuit and bit 6 represents the faulted circuit. Prior to simulation, the control program makes an alteration to the compiled simulator. The instruction that loads the value from GATE_TABLE into register AX is replaced by a call to a subroutine. The subroutine loads the value from GATE_TABLE into register AX and then performs an AND operation on that value using the 8-bit mask 10111111. The subroutine then returns to the compiled simulator.

This method of bugging the model has the effect of causing the OR gate to always receive a 0 on its upper input, regardless of what value is generated by AND gate *G*. Suppose $A = B = C = 1$ and $D = E = F = 0$. Inputs *A*, *B,* and *C* are assigned an 8-bit vector consisting of all-1s, while *D*, *E,* and *F* are assigned vectors consisting of all-0s. During simulation the good circuit, bit 7, will simulate the OR gate with input values (1,0,0) and the circuit corresponding to bit 6 will simulate the OR with input values (0,0,0). As a result, bit positions 7 and 6 of the result vector will receive different values at the output of gate *I*. ■■

In practice, the bugging operation can use seven bits of the byte. In the example just described, bit 5 could represent the fault corresponding to the center input of gate *I* SA0. Then, when the program loads the value from GATE_TABLE+2 into register BX, it again calls a subroutine. In this instance it applies the mask 11011111 to the contents of register BX, forcing the value from gate *H* to always be 0, regardless of what value was computed for *H*. When bugging a gate output, the value is masked before being stored in GATE_TABLE. If modeling a SA1 fault on an input, the program performs an OR instruction using a mask containing 0s in all bit positions except the one corresponding to the faulted circuit, where it would use a 1.

In a combinational circuit or a fully synchronous sequential circuit, one pass through the simulator is sufficient to obtain fault simulation results. In an asynchronous sequential circuit it is possible that the fault-free circuit or one or more of the faulty circuits is oscillating. In a compiled model in which feedback lines are represented by pseudo-outputs and corresponding pseudo-inputs (see Section 2.6.2), oscillations would be represented by differences in the values on pseudo-outputs and corresponding pseudo-inputs. In this case it would be necessary to run additional passes through the simulator in order to either (a) get stable values on the feedback lines or (b) deduce that one or more of the circuits is oscillating.

At the end of a simulation cycle for a given input vector, entries in the circuit value table that correspond to circuit outputs are checked by the control program. Values in bit positions [6:0] that differ from bit 7, the good circuit output, indicate detected faults—that is, faults whose output response is different from the good circuit response. However, before claiming that the fault is detected by the input pattern, the differing values must be examined further. If the good circuit response is X and the faulted circuit responds with a 0 or 1, detection of that fault cannot be claimed.

3.6.2 Performance Enhancements

In the 80×86 program, when performing byte-wide operations, parallel simulation can be performed on the good circuit and seven faulted circuits simultaneously. In general, the number of faults that can be simulated in parallel is a function of the host computer architecture. A more efficient implementation of the parallel fault simulator would use 32-bit operations, permitting fault simulation of 31 faults in the time that the byte-wide fault simulator fault simulated 7 faults. Members of the IBM mainframe family, which are able to perform logic operations in a storage-to-storage mode, can process several hundred faulted circuits in parallel.

Regardless of circuit architecture, a reasonable-sized circuit will contain more faults than can be simulated in parallel. Therefore, numerous passes through the simulator will be required. On each pass a fault-free copy of the simulator is obtained and bugged. The number of passes is equal to the total number of faults to be simulated divided by the number of faults that can be simulated in a single pass. It is interesting to note that although we adhere to the single-fault assumption, it is relatively easy to bug the simulator to permit multiple-fault simulation.

The compiled simulator is memory efficient. Augmented with just a circuit value table and a small control program, the compiled simulator can simulate very large circuits. Simulation time is influenced by three factors:

The number of elements in the circuit

The number of faults in the fault list

The number of vectors

As the circuit size grows, the size of the compiled simulator grows, and, because there are more elements, there will be more faults; therefore more fault simulation passes are necessary. Finally, more vectors are usually required because of the increased number of faults. As a result of these three factors, simulation time can grow in proportion to the third power of circuit size, although in practice the degradation in performance is seldom that severe.

A number of techniques are used to reduce simulation time. Most important are the concepts of fault dominance and fault equivalence, which remove faults that do not add information during simulation (cf. Section 3.4.5). Simulation time can be reduced through the use of stimulus bypass and the sensitivity list (cf. Section 2.7). These techniques avoid the execution of code when activity in that code is not possible.

Circuit partitioning can be useful in reducing simulation time, depending on the circuit. If the subcircuits that drive two distinct sets of outputs have very few gates in common, then it becomes more efficient to simulate them as separate circuits. The faults that occur in only one of the two subcircuits will not necessitate simulation of elements contained only in the other subcircuit. Circuit partitioning can be accomplished by backtracing from a primary output as follows:

1. Select a primary output.
2. Put gates that drive the primary output onto a stack.
3. Select an unmarked gate from the stack and mark it.
4. Put its unmarked driving gates onto the stack.
5. If there are any unmarked entries on the stack, go back to step 3.

The gates on the stack constitute a subcircuit, called a *cone*, which can be processed as a single entity. Where two subsets of outputs define nearly disjoint circuits of approximately the same size, the simulator for each circuit is about half its former size; there are half as many faults, hence perhaps as few as half as many vectors for each circuit. Thus, total fault simulation time could decrease by half or more.

A practice called *fault dropping* is used to speed up fault simulation performance. The simulator drops faults from the fault list and no longer simulates them after they have been detected. Continued simulation of detected faults can be useful for diagnostic purposes, as we shall see later, but it requires additional simulation time. Many faults, perhaps as many as half or more, are detected quite early in the simulation, within the first 10% of the applied test vectors. By dropping those faults, the number of passes through the fault simulator for each vector is significantly reduced.

States applied analysis[4] employs logic simulation to determine which faults are detectable by a given set of test vectors. During fault simulation, an AND gate is evaluated to determine if stuck-at-1 faults are detectable at its inputs. To detect a fault on an input to an AND gate, it is necessary to have a 0 on the faulted input and logic 1s on all other inputs. With that combination, a fault-free gate responds with a 0 at its output, and a gate with a stuck-at-1 fault on that input responds with a 1 at its output. An analogous consideration applies to the OR gate. If, for a complete set of test vectors, an n-input AND gate never receives an input stimulus consisting of a 0 on input i and 1s on the remaining $n - 1$ inputs, then the stuck-at-1 fault on input i will never be sensitized. Since the fault is not sensitized, it is pointless to fault simulate that fault.

3.6.3 Parallel Pattern Single Fault Propagation

Parallel fault simulation uses the extra bits in a word to fault simulate $n - 1$ faults in parallel, where n is the word size or register size of the host computer. Parallel pattern single fault propagation (PPSFP) can be thought of as being orthogonal to parallel fault simulation.[5] Each bit in a computer word represents a distinct vector. The fault-free circuit is first simulated and the response at the output pins is recorded for

that vector. Then, given a host computer with an n-bit wide data path, n vectors are simulated in parallel. However, only one fault is considered, and the circuit is combinational.

Consider again the circuit of Figure 3.10. For the sake of illustration, assume that we are going to apply all 64 possible input combinations to the six inputs. We would start by applying 32 vectors to the fault-free circuit. Since we are going to apply all combinations, we could simply create a truth table for the six values. Then, for the first 32 vectors, the simulation values would be

```
A = 01010101010101010101010101010101
B = 00110011001100110011001100110011
C = 00001111000011110000111100001111
D = 00000000111111110000000011111111
E = 00000000000000001111111111111111
F = 00000000000000000000000000000000
```

In this matrix, the leftmost column represents the first vector, the second column represents the second vector, and the remaining columns are interpreted likewise. The first row is the sequence of values applied to primary input A by each of the 32 vectors, the second row is applied to input B, and so on. As a result, this matrix causes logic 0 to be applied to all inputs on the first vector, and on the second vector the value on input A changes from 0 to 1. When simulating the fault-free circuit, the simulation begins, as before, by ANDing together the values representing inputs A and B. That is followed by ANDing C and D, then complementing the result.The remaining operations are determined similarly. The result is

$$00010001000100010001000100010001 = AB = G$$
$$11111111111100001111111111110000 = \overline{CD} = H$$
$$00000000000000000000000000000000 = EF = J$$
$$11111111111100011111111111111111 = AB + \overline{CD} + E = I$$
$$11111111111100011111111111111111 = K$$

Vector K represents the fault-free response of the circuit for each of the 32 vectors. To get the circuit response for a stuck-at-0 fault on the input to gate I driven by gate G, replace the response vector AB by the all-0 vector and resimulate. The result is

$$11111111111100001111111111111111 = K$$

Note that, counting the leftmost bit as position 31, bit 16 is 0, where it had previously been a logic 1. Hence, we conclude that the vector $A,B,C,D,E,F = 111100$ will detect a stuck-at-0 on the input to gate I that is driven by gate G.

In a much larger, more realistic circuit, made up of tens or hundreds of thousands of gates, it is inefficient to simulate all of the gates. Rather, fault simulation can begin at the point where the fault occurs, and proceed forward toward the outputs. If the circuit is rank-ordered, then no element is evaluated until all of its predecessors

are simulated, so the correct values will already have been computed during simulation of the fault-free circuit. For the faulted gate, the vector representing the values on the input or output that is faulted is modified to represent the stuck-at value for all of the applied vectors.

If a compiled fault simulator is used, a jump can be made into the compiled netlist at the point where the fault exists. A table-driven simulator can simply pick up the values at the fault origin and propagate logic events forward (recall that an event is a signal change). Since, in combinational circuits it is not uncommon for a high percentage of stuck-at faults, perhaps 50% or more, to be detected within the first 32 vectors, many faults will only require one pass through the simulator. Further savings can be realized on a circuit with many output pins by halting simulation as soon as an error signal reaches any output pin.

3.7 CONCURRENT FAULT SIMULATION

It should be clear by now that the purpose of fault simulation is to evaluate the effectiveness of a set of input vectors for detecting stuck-at faults in a circuit. The fault simulator does this by determining whether or not the set of vectors establishes a path from the point where the fault originates to one or more output pins, such that the good circuit and faulted circuit respond differently all along that path. In addition, the parallel fault simulation algorithms use the host computer resources to process either n faults in parallel or n vectors in parallel.

The concurrent fault simulation algorithm is capable of simulating n faults simultaneously, where n may represent one fault or it may represent several thousand faults.[6] Records are kept for each fault as it causes error signals to occur. When the error signal is blocked, or prevented from propagating further in the circuit, no additional records are generated for that fault. The number of faults, n, that can be simulated concurrently is limited only by the amount of memory available. We begin by examining the underlying concepts of concurrent fault simulation in detail for the case where n is one and then describe the concurrent fault simulation algorithm more formally.

3.7.1 An Example of Concurrent Simulation

The circuit in Figure 3.11 will be used to illustrate concurrent fault simulation. Assume the presence of a stuck-at-1 fault on the top input to gate H. The circuit will first be analyzed without the stuck-at fault. The circuit is annotated with logic 1s and 0s. With the values indicated, the 1 at primary input C is inverted by F to become a 0 at the input to H. That, in turn, causes the output of H to become a 1. However, the signal cannot propagate because the 0 from G is a blocking signal at J and the 1 at primary input E is a blocking signal at K. A second vector is now applied in which the value of A switches to a 0. This causes the output of G to switch to a 1. That, in turn, causes the output of J to switch to a 1.

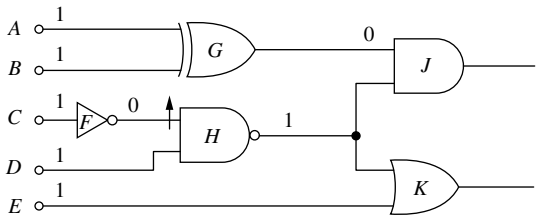


Figure 3.11 Simulating small changes.

Now consider what happens when the top input to gate *H* is SA1. In the presence of the fault, *H* simply inverts the signal at input *D*. With a 1 at the *D* input, the output of *H* is a 0. As in the previous case, signal paths through both *J* and *K* are blocked during the first vector. On the second vector, *G* switches to a 1 and the signal from *H* is now enabled through the bottom input to *J*. However, the output of *H* is now a 0 because of the fault, so the output of *J* fails to switch, it remains a 0.

The stuck-at fault on the input to *H* affected only the signal path connecting *H* to *J* and *K*, and the output response at *J*. Furthermore, the effect of the fault was visible at an output only on the second vector. During the first vector the fault response from *H* propagated to *J* and *K*, but the blocking signals *J* and *K* prevented the signal from propagating to the output.

In this small circuit a fault affected a significant part of its behavior. In real circuits a fault may affect less than one percent of the circuit values. In such circumstances it makes no sense to simulate the entire faulted circuit. The simulator is more efficient if it only keeps track of those signals that are affected by the fault. To do so, it must have a way to record the circuit faults, and it must have a way to record circuit values that are affected by the faults. This can be done by allocating a field to represent fault type in the data structures that represent the circuit topology.

For example, the data structure for an *n*-input AND gate may have a special code to represent each of its inputs SA1. Another code might indicate a SA0 on the output. Additional codes can be used to represent shorts across adjacent pins, or internal faults that can only be detected by special combinations on the inputs—for example, 0s on two or more inputs. Then, during simulation, the simulator checks the input values at the gate currently being processed to determine if they cause any of the faults at that gate to become sensitized. If a fault becomes sensitized, its effects are propagated forward. This tremendous flexibility in modeling defects is one of the major attractions of the concurrent fault simulator.

To propagate the effects of the fault, it is necessary to record all signal values that differ from the values in the fault-free circuit wherever they occur. These can be recorded using a flag to indicate that a particular element or net has values for the faulted circuit that differ from the values computed for the original circuit. In many cases the original circuit and the faulted circuit can be simulated simultaneously. For example, on the first vector, the inverter produced a 0 at the input to *H*, whereas the faulted circuit has a constant 1 at that input.

Now, when simulating gate *H*, its output produces a 1 for the original circuit and a 0 for the faulted circuit, and these signals can be propagated simultaneously. But, what happens when the value on input pin *D* is 0 for a particular vector? The output of *H* is then a 1 regardless of what value appears at its upper input. If *D* changes to a 1 on the next vector, the original circuit retains a 1 at the output of *H*, but in the faulted circuit *H* switches to 0. The simulator must be able to propagate this event for the faulted circuit without corrupting the value existing in the original circuit.

3.7.2 The Concurrent Fault Simulation Algorithm

The operations described in the preceding subsection will be formalized; but before doing so, it will be helpful to briefly review and summarize the operations that took place. First, all differences between the original and modified circuits were explicitly identified. Although a stuck-at fault was assumed, the analysis could just as easily have been describing a design change, wherein we wanted to contrast circuit behavior with and without the inverter labeled *F*. Then, two situations were identified for which it would be necessary to evaluate signals in the faulty circuit:

1. Whenever an event occurred in the original circuit for which a different signal occurred in the faulted circuit.
2. Whenever an event in the original circuit did not propagate to the gate output, but caused a signal in the faulted circuit to propagate to the gate output and beyond—for example, the change at the output of gate *G*.

It was not obvious in this small circuit, but the error signal for the faulty circuit could, in this second case, spread throughout the circuit and cause many hundreds or thousands of differences. For example, if a fault caused the wrong function to be selected in an ALU, over half of the gates in the ALU array could have incorrect logic values.

Concurrent fault simulation is essentially a data processing task. Its purpose is to record data that identify differences in simulation response between two or more circuits. While it can be used to distinguish differences between virtually any two circuits, its primary purpose is to compute fault coverage for test programs. The differences that it records are those between the fault-free circuit and one or more (usually many more) faulty circuits that are very similar to the fault-free circuit, differing only in that each of the faulty circuits represents a different fault. The goal is to determine, for each of the faulty circuits, whether or not the effects of the modeled faults are observable at a primary output where they can be detected by a tester.

To perform a concurrent fault simulation, it is necessary to define data structures that record simulation differences between the circuits. However, first it must be decided which differences are important. For example, one piece of information that must be permanently maintained throughout simulation is the source, or location, of defects for each of the faulted copies of the circuit. Another piece of information is the value of error signals generated for each of the defects. When an error signal arrives at a gate, it is also necessary to identify which pin or pins receive the error signal.

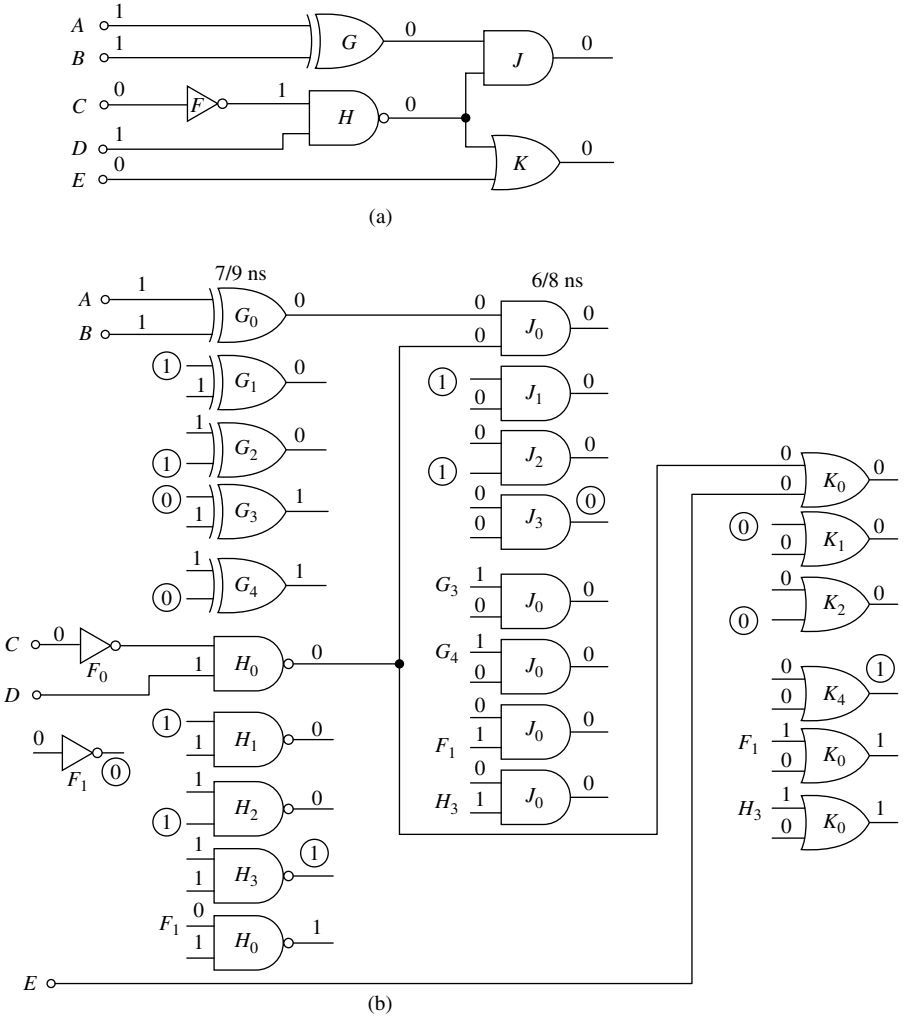


Figure 3.12 (a) Circuit for concurrent fault simulation. (b) Circuit with linked fault effects.

Recording information in the concurrent fault simulator is accomplished by appending or linking new copies of a circuit element to the original element. These copies appear wherever faults cause signal values in a circuit to differ from good circuit signals. Furthermore, new circuit elements are added for as long as the error signal continues to propagate. This is illustrated conceptually in Figure 3.12. In (a) the fault-free circuit is illustrated with correct logic values at each net. In (b) a modified version is illustrated in which each of the gates is replicated several times. In the following discussion, the element X is followed by the subscript *i*, which is interpreted as follows:

0	fault-free circuit
1	input 1 SAX
...	
n	input n SAX
n + 1	output SA0
n + 2	output SA1

where the element X is assumed to have n inputs and SAX denotes SA1 for an AND gate, SA0 for an OR gate.

The purpose of the multiple copies of the various gates is to simultaneously represent the fault-free gate and instances of the gate where either faults originate or the logic value at the input of the gate is affected by faults occurring at other gates. The concurrent fault simulation algorithm recognizes two classes of faults, namely, fault origins and fault effects. A *fault origin* (FO) is a gate at which a fault originates. An input fault origin (IFO) occurs on a gate input, and an output fault origin (OFO) occurs on the output. Fault origins are linked together and attached to the unfaulted gate. A separate FO is used for each fault.

If an FO causes the input value at a destination gate to differ from that of the fault-free gate, then a *fault effect* (FE) is created or *diverged* and attached to the fault list of the destination gate. Whenever the output value of an FO or FE is different from that of the corresponding unfaulted circuit, the FE or FO is said to be *visible*. When the output of an FE or FO becomes visible, an FE is diverged at the destination gate. FEs continue to be diverged forward in the circuit until either the error signal is no longer visible or a primary output is encountered. When the error signal is no longer visible, the FE is *converged*.[7]

These concepts are illustrated in Figure 3.12(b). Note first that there are five copies of gate G. The copy G_0, driven by inputs A and B, corresponds to the fault-free circuit. The remaining four copies are all IFOs. Copy G_1 (G_3) has one input SA1 (SA0) and the other input driven by input B. Copy G_2 (G_4) has one input SA1 (SA0) and the other input driven by input A. There are two copies of gate F, one corresponding to the fault-free circuit and an OFO corresponding to the output SA0. Gate H has a fault-free copy H_0 and IFOs for SA1 faults on each of its inputs as well as an OFO for a SA1 fault on its output. It also has an FE, which consists of unfaulted copy H_0 driven by fault origin F_1. Gates J and K also have several copies which are interpreted similarly.

The circled logic values in the figure are used to denote signals that are SA1 or SA0; hence the gate at which they occur are IFOs or OFOs. FEs are indicated by an unfaulted copy of a gate in which one or more inputs are sourced by an FO or FE. In the discussion that follows, the notation X_0/Y_i represents a fault effect that originates at fault origin Y_i and is diverged at gate X to drive an unfaulted copy X_0 of X. The rise and fall delays for the elements are indicated above the unfaulted copy of the elements.

Before describing the rules for concurrent fault simulation, we informally describe what happens when an event occurs. Given the signal conditions and the attached

fault effects indicated in Figure 3.12(b), suppose that primary input D changes to 0. It drives not only the unfaulted circuit H_0 but also some copies, including H_1 and the fault effect H_0/F_1. Fault origin H_2 is unaffected by the event because the gate input connected to primary input D is stuck-at-1. The OFOs H_3 and H_4 are unaffected by any input change. The gate H_0 in the unfaulted circuit must be simulated. The corresponding gates H_1 and H_0/F_1 in the faulted circuit must also be simulated.

When H_0 is simulated, its output switches from 0 to 1, therefore it must be scheduled for processing at time $t + 4$. Gate H_1 also changes but the value on H_0/F_1 does not change; therefore H_1 is scheduled but H_0/F_1 is dropped from further processing. Gates H_0 and H_1 are retrieved from the scheduler at time $t + 4$ and their outputs are updated. Fault lists attached to gates in the fanout of gate H_0 are processed. We describe here only the processing for gate J_0. Fault effects H_3 and H_0/F_1 no longer differ from H_0, so they are converged and dropped from the fault list attached to J_0. However, H_2 and H_3 now differ from H_0, so those fault signals must be linked to the fault list attached to J_0; that is, they are diverged at J_0. Also, the change on H_0 reaches the lower input of FEs J_0/G_3 and J_0/G_4, so those FEs must be simulated. Since the outputs of those FEs change, they must be placed on the scheduler.

The fault origin H_1 was also simulated. Its output is identical to that of the unfaulted copy. A check of the fault list attached to J_0 shows that there is no fault effect labeled H_1 in the list, so no further processing need take place. Those fault effects that eventually reach a primary output—in this case J_4, J_0/G_3 and J_0/G_4—define a sensitized path from the fault origin to the output; hence they correspond to detected faults.

It is possible that the faulted copy changes and the unfaulted copy does not change. For example, if the change on input D is followed by a change on input C, then H_2 will change while H_0 remains unchanged. In that case, it is necessary to trace the faulted output change to the destination gate(s) and perform divergence and convergence, as the situation warrants. It is also possible that the unfaulted copy may change in one direction while the faulted copy changes in the opposite direction, as would be the case when primary input A changes. G_0 and G_2 change to 1, G_4 changes to 0, and G_1 and G_3 are unaffected. Furthermore, because the rise and fall times for G are different, G_0 and G_4 are placed in different time slots on the scheduler.

This model expands and contracts as input signals change. The basic fault-free circuit remains fixed, but the remainder of the circuit is quite fluid. Gates with fault signals are added when fault effects cause the value on a gate input to differ from the corresponding value on the good circuit. Gates in the fanout of a faulted element continue to exist as long as the error signal persists. If the logic values on a gate change so that an error signal is no longer distinguishable from the fault-free signal, then that path terminates. When an error signal terminates, its forward propagation path must be deleted in its entirety.

Implementation of the concurrent fault simulator does not require complete descriptor cells for each fault signal that differs from the good circuit signal. Rather, an abbreviated descriptor cell (ADC) is used for FEs and FOs, since much of the information required by the simulator for the purpose of evaluation is identical for faulted and fault-free circuits. A typical format for the ADC is illustrated in

Figure 3.13. The fault-free cell and all related faulted cells are linked via pointers. With the exception of the ADC, FOs and FEs are similar to regular gates. They use the same functions as fault-free elements to schedule and evaluate elements. However, events on FOs and FEs can only affect FEs with the same identification number, whereas the signal from the good gate affects both the fault-free circuit and all faulted circuits. The receiving pin number and the input states are needed to compute the behavior of the element with the error signal and contrast it with the response of the fault-free element. To help expedite processing, ADCs can be ordered by fault identification number when linked to a descriptor cell.

When a logic change occurs on the output of a gate in a fault-free circuit, processing for an FO or an FE depends on whether it is linked to the fault list for the source gate, called the emission list (ELIST), or the fault list for the destination gate, called the receive list (RLIST), or both. The rules are as follows:

If in ELIST only: Diverge a copy (an FE) of the destination gate with input states identical to those that existed on the unfaulted destination gate before the change arrived.

If in RLIST only: If it is an OFO, no action is taken. If it is an IFO, simulate unless the input change occurred on the faulted input. If an FE, simulate with the same change that occurred on the good gate.

If in both: If the FE or FO output value in ELIST is X, then take the same action as when the FE or FO is in RLIST only. Otherwise, compare the input states of the FE in the RLIST to the states on the unfaulted gate and converge if they are identical.

Example The events that occur when input D changes from 1 to 0 are described again. The event at D is applied to the input of H_0 and simulated. Because its output changes, H_0 is scheduled for processing in time slot t + 4. After H_0 is scheduled, its attached fault list is processed. No faults were attached to primary input D, so there is no ELIST; hence the "in RLIST only" rule is used. H_1 and H_2 are IFOs, so H_1 is simulated but H_2 is not simulated. H_3 is an OFO; therefore no action is taken. H_0/F_1 is an FE so it is simulated with the same event that occurred on the unfaulted gate.

When H_0 is retrieved from the scheduler, gates J_0 and K_0 are simulated. However, only the processing for J_0 is described here. The output of gate J_0 did not change; nevertheless, the fault list attached to J_0 must be processed. J_1 is simulated and its output changes, so it must be scheduled. J_2 is faulted on the input that changed, so no processing is required. J_3 and J_4 are OFOs, so they are not processed. Fault effects G_3 and G_4 are in the RLIST but not in the ELIST for H_0, so they are simulated and placed on the scheduler.

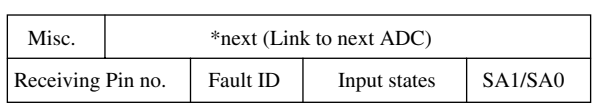


Figure 3.13 Abbreviated descriptor cell.

There are two FOs, H_2 and H_3, in the ELIST of H_0 that differ from H_0 and are not in the RLIST, so it is necessary to diverge FEs J_0/H_2 and J_0/H_3 with input values identical to the values on J_0 before the change arrived. There are two FEs, J_0/F_1 and J_0/H_3, that are in both the ELIST and the RLIST. The logic values on the inputs of J_0/F_1 and J_0/H_3 are identical to the values on the inputs of J_0 after the event arrived from H; therefore the two FEs are converged. ■■

Events originating in the good circuit can affect good circuits and possibly all faulted circuits, according to the rules given above. However, events generated by a fault circuit can only affect faulted circuits with the same fault ID. Therefore, when the output of H_1 changed, the only fault IDs that it will affect are those labeled H_1 in the fault list attached to J and K. Since there are none and since the output of H_1 remains identical to the value on the unfaulted circuit H_0, no further processing is required.

3.7.3 Concurrent Fault Simulation: Further Considerations

Concurrent fault simulation was explained using the rather simple circuit of Figure 3.11. That circuit had simple logic elements, including AND, OR, and XOR gates. To fully appreciate the concurrent fault simulation algorithm, it is important to realize that its operation is not materially affected by the types of elements in the circuit. Apart from the processing required to cope with divergence and convergence of fault origins and fault effects, in other respects the processing of these short-lived fault elements is identical to the processing of the more permanent good circuit elements. Fault modeling capabilities are far more flexible than for other fault simulation algorithms because a faulted model can represent a delay fault or virtually any other fault for which modeling code can be written.

Latches and flip-flops are processed in a manner similar to the logic elements. In fact, user defined primitives (UDPs) found in many Verilog designs, as well as RTL models, can be processed just like logic elements. A major problem with UDPs and RTL models is the fact that granularity can be quite coarse. A UDP, even if it is strictly combinational, may contain reconvergent logic, hence stuck-at faults on the inputs of the UDP may not represent all possible internal stuck-fault modes. If an RTL model has storage elements, the state of one or more of these elements may be affected by an error signal entering the model. It is necessary to recognize that the state is affected and the states for all error signals must be recorded, just as states for logic gates are recorded.

If an RTL module has many sequential elements, fault processing may be accomplished by diverging individual copies of the RTL block for every fault that appears at its inputs, as well as for every fault that causes one or more of its internal storage elements to assume an incorrect value. This can require a massive amount of memory. An alternative approach, which may provide faster processing speed and more efficient memory utilization, would be to create submodules for every latch or flip-flop in the RTL module. Then, if a fault effect causes one or more of these flip-flops or latches to assume an incorrect value, link lists of fault effects can be linked to them just as they would if they were primitive gate-level elements. It would not be

necessary to create an entire RTL block for a fault that affected only a single flip-flop within the RTL module. The FEs that affected only a single flip-flop would only be linked to that flip-flop.

When simulating sequential circuits, faults can cause a circuit to enter an incorrect circuit state and remain there for an indefinite period. A register may be loaded from a bus, and that value may be held for many hundreds or thousands of clock cycles, without being used. Finally, the value may be read by some other functional unit, and the error signal may propagate forward and eventually be detected at an output pin. If it is necessary to diagnose the source of an error at an output pin, it may require some careful analysis to build a causal link back to the fault origin.

Efficient memory management is critical to good performance when performing concurrent fault simulation. Virtual memory management is often used by operating systems in order to share main memory among different jobs, but it is not practical for concurrent fault simulation. The simulation run will simply thrash. If a run requires more main memory that is available on the host system, the fault simulator should split the fault list into two or more partitions and run them individually.

It is interesting to note that splitting the fault list can sometimes improve performance even in cases where there is sufficient memory to perform the simulation in a single pass through the fault simulator. This occurs because the fault simulator is processing linked lists of fault effects; and as the fault list increases, these link lists grow in length, with the result that traversing these link lists begins to seriously impact performance. The number of passes is estimated based on circuit size, fault list size, the amount of available memory, and the amount of memory used to implement the descriptor cells and abbreviated descriptor cells. Since some of the numbers are dependent on the implementation, they must be derived empirically.

A concurrent fault simulator will sometimes classify a fault as hypertrophic. A *hypertrophic fault* spreads throughout a circuit and causes FEs to be linked to a great many logic elements. An earlier paragraph described a fault in control logic that caused the wrong function in an ALU to be performed. If an OR operation was supposed to be performed, but a fault causes a subtract operation to be performed, then conceivably half or more of the logic signals in the ALU could be incorrect. Sometimes a concurrent fault simulator will drop a hypertrophic fault on the assumption that a fault so pervasive will inevitably cause an FE to reach an output and become detected. A *hyperactive fault* is one that causes a large number of evaluations. Sometimes a fault can cause oscillations in a circuit. This is an especially serious problem if a zero-delay loop is oscillating because the scheduler cannot advance time until the oscillation is resolved. The oscillating signals can be set to X, or the fault origin can be deleted.

3.8 DELAY FAULT SIMULATION

The emergence of deep submicron technology (DSM) has brought ever faster ICs. It has also brought a growing vulnerability to delay faults—that is, manufacturing imperfections that cause a device to fail to operate correctly at its intended clock

speed—even though it may be functionally correct. Defects that would not have affected performance in a previous generation device suddenly induce erratic behavior. It may not be a solid defect, such as an open, or a short between two metal runs on an IC. Rather, it might be a wire run with too much resistance, capacitance, or loading, which manifests itself as excessive propagation delay, either at room temperature or at the low or high end of the operating spectrum. For example, ICs intended for the automotive market have to operate correctly at temperatures up to 120°F in the Arizona desert, and down to −50°F in the upper midwest and Canada.

As a result of these operating extremes, it has become increasingly important to develop tests for critical paths—that is, those paths with the greatest delay from a source to a destination. The source may be either a primary input or the output of a flip-flop, while the destination may be a primary output or the input of another flip-flop. This is illustrated in Figure 3.14. Rising edges emanate from U1 and U2. These signals result from logic 1s on the inputs of U1 and U2 being clocked through the flip-flops and replacing 0s on their outputs. The rising edge from U1 passes through some combinational logic, indicated by the pair of wavy lines, and reaches U3 as a rising edge. The edge from U2 reaches U4 after experiencing an odd number of inversions. The rising edge is blocked on its way to U5, perhaps because it had to pass through an AND gate whose other input is the blocking 0 value.

It was pointed out in Section 3.7.1 that the concurrent fault simulator is well-suited to modeling many types of faults. Among those that it is well-suited to handling is edge propagation. Whenever the value on the input of a flip-flop is the complement of the value on its output, an edge emanates from the flip-flop on the next active clock edge. A fault-effect (FE) can be diverged from that flip-flop which can be processed in a manner analogous to the way in which FEs are processed for stuck-at faults. If the FE representing the edge (an edge FE) reaches the input of one or more destination flip-flops, it becomes trapped in that flip-flop.

Referring again to Figure 3.14, the input to U3 is an edge that originated at U1. If the circuit is working correctly, a 1 is clocked into U3 during operation. If there is a delay fault, the 1 fails to reach U3 before the next clock edge and a 0 gets clocked into U3. This is represented by the 1/0 at the output of U3, which represents 1 on the good circuit and 0 on the faulty circuit. Once a delay fault has been clocked in, it can be treated like a stuck-at fault at the destination flip-flop. Propagation of the FE from that point can be performed exactly as it is performed for stuck-at faults. If the FE reaches an output, the tester can determine whether the delay fault affected U3.

Once an edge FE becomes trapped, it continues to exist until it either reaches an output or converges. However, the FEs representing edges are removed at the end of each clock period by a garbage collection routine. Another delay FE does not appear at the flip-flop until once again the input and output of the flip-flop are complements of one another. This is analogous to the fault origin (FO) for stuck-at faults. Note that it is possible for an edge FE to initially becomes blocked at an AND gate or an OR gate. Suppose an edge FE reaches a 2-input AND gate which has a 0 on its other input. That other input may change from 0 to 1 after the edge FE arrives. In that case, the edge FE should remain converged, because there is another path of longer duration than the path from U1 to U3.

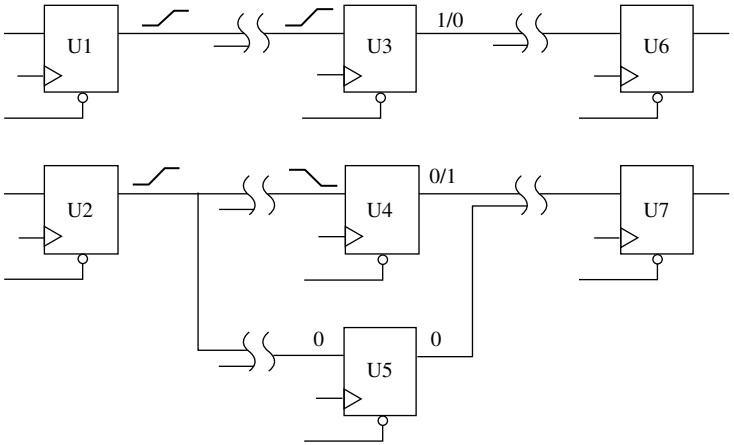


Figure 3.14 Delay fault propagation.

The abbreviated descriptor cell, Figure 3.13, is slightly modified in order to reflect that fact that the FE represents an edge rather than a stuck-at fault. The fault ID has to be expanded in order to identify the source and destination of the edge. A postprocessor can then use the fault IDs to identify all paths that have been exercised by the test. The user can inspect the report to determine if the most critical paths have been exercised. The delay fault simulation capability is easily integrated into an existing concurrent fault simulator with very little effort. Of course the effectiveness of edge fault simulation depends totally on the effectiveness of the vectors that are evaluated. In Chapter 7 we examine methods for generating test vectors directed at delay faults.

3.9 DIFFERENTIAL FAULT SIMULATION

The differential fault simulation (DSIM) algorithm described here, so called because of its use of the differences between any two circuits, is based on the assumption that the circuit being fault simulated is synchronous and that all circuit elements have zero delay. These assumptions are not unlike those on which parallel fault simulation and PPSFP fault simulation are based. However, DSIM goes beyond them in that it retains state information from one vector to the next; hence it can be applied to sequential circuits.[8] In that respect, it bears a resemblance to the concurrent fault simulation algorithm.

DSIM will be described with the help of some notation. The term $B_{i,j}$ denotes the circuit status for the ith fault and the jth vector. The circuit state for faulty circuit $B_{i+1,j}$ is derived from faulty circuit $B_{i,j}$ by simulating the differences of their fault origins as the initial fault events. The circuit corresponding to $i = 0$ is the fault-free circuit. The circuit state for $B_{0,\mathrm{j}}$ is obtained by performing a logic simulation of the

inputs for the *j*th vector. Note that when simulating a sequential circuit, there are usually state differences at the storage elements, and these must also be evaluated. The algorithm for DSIM follows:

```
for(i = 0; i < no_vectors; i = i+1) {
  if(i == 0)                  // first vector
    initialize circuit;       // set all nodes to X
  else {
    remove previous injected fault; // fault-site event
                                   //  source
    restore current states;  // state-difference event
                             // source
    apply primary input values; // input-difference event
                               // source
    perform event-driven simulation;
    record next-state differences;
    store primary output values;
    sensitized_output_counter = 0;
    for (all undetected faults) {
      remove previous injected fault; // fault-site
                                     // event source
      inject current fault;
      recover current states;// state-difference event
                             // source
      perform event-driven simulation;
      record next-state differences;
      if (sensitized_output_counter > 0) // FE reached
                                        // output pin
        drop the fault;
    }
  }
}
```

The general approach in DSIM is to define events that must be propagated forward to the outputs. For the fault-free circuit, events on primary inputs are referred to as *input difference event sources*. For faulted circuits, both the previously injected fault, which is removed, and the current fault, which is injected, are referred to as *fault site event sources*. Regardless of whether the event is an input event or a fault event, the operation is essentially the same: Establish the initial events and then perform event-driven simulation from the point where the event originated, until either a primary output or a memory element is reached, or the events converge. If a fault event reaches an output, an output counter is adjusted. After simulation of each faulty circuit, if the counter has a nonzero value, the fault is detected.

Since error signals are only recorded at memory elements, the amount of memory required to retain a history of each fault is considerably less than that required for concurrent fault simulation. However, the fact that error signals are stored at memory elements implies that all memory elements must be explicitly identified. If all storage elements are modeled as latch or flip-flop primitives, it becomes trivial to identify them. However, if there are storage elements defined by feedback created by logic primitives, such as cross-coupled NAND gates, or, worse still, more complex feedback configurations, this may cause DSIM to compute erroneous results.

3.10 DEDUCTIVE FAULT SIMULATION

Deductive fault simulation[9] simulates only the fault-free circuit. The simulator deduces which faults are tested by each input vector and creates lists of those that are sensitized at each node. In some respects it is analogous to concurrent fault simulation. As simulation proceeds, some faults cease to be sensitized, their effects become blocked, and they are dropped by the simulator. Meanwhile, other faults become sensitized and are added to the list of sensitized faults.

To illustrate, consider the fault-propagating characteristics of a three-input OR gate. Associated with each input is a list of faults from preceding logic that are sensitized up to the input of the OR gate. If the present values on the OR gate inputs are all 0s, then the fault list on the output of the OR gate is the union of the fault lists on all the inputs. This follows from the fact that the fault list on any input is the set of faults that cause that input to assume a value that is opposite to its correct value. Conversely, if the fault-free signals at all three nodes are 1s, then a fault symptom could propagate through the OR gate only if it could cause all three inputs to assume incorrect values. Therefore, the set of faults that propagates to the output of the OR gate is the set that results from the intersection of the fault lists at the three inputs. If one or two inputs are at 1 and the other is at 0, then the computations get slightly more complex.

Example Assume an OR gate for which the fault lists are:

$$\text{A} = \{1,2,4,7,11\}$$

$$\text{B} = \{2,5,7,8\}$$

$$\text{C} = \{1,3,7,12\}$$

If all three input values are 0, then the output fault list is $D = A \cup B \cup C \cup \{d_1\}$ where d_1 represents a SA1 on the OR gate output. For the sets A, B, and C listed above, $D = \{1,2,3,4,5,7,8,11,12,d_1\}$. If all three inputs are at logic 1, then the output fault list is the set $D = A \cap B \cap C + \{d_0\}$ where $\cap$ denotes set intersection and $\{d_0\}$ denotes the output SA0. In this example, $D = \{7, d_0\}$. If the upper two inputs are logic 1s and the lower input is a 0, then the only way to get an incorrect output is if a fault f changes the values of the upper two inputs but does not change the lower

output—that is, if $f \in A \cap B - C$. In this example, fault 2 fits that requirement; hence it will propagate to the output if the OR inputs are {1,1,0}. To that intersection the output fault d_0 is added. The result is $D = \{2, d_0\}$. If any single input is at 1, then that input SA0 will also propagate to the output and must be added to the list.

A general rule for processing OR gates follows:

- To the fault list at each input, add the fault corresponding to that input SA0 if the value on that input is a 1,
- If all inputs are 0, then form the union of all these sets and add the fault corresponding to the output SA1.
- If one or more inputs are 1, then
  - Form the intersection S of sets corresponding to inputs that have 1s.
  - Form the union T of sets corresponding to inputs that have 0 values.
  - Compute $S - T$.
  - Add the fault corresponding to the output SA0. ■■

Deductive fault simulation can require processing enormous lists of faults using equations for manipulation of these lists which vary according to the values on the inputs of the gate being processed. In an event-driven environment, extensive list processing may be required even when no logic activity occurs. For example, if the three input OR gate has values (1,1,0) on its inputs and if the inputs change to (1,0,0) in response to a logic change, then the formula for computing the output fault list changes; hence the output fault list for the gate must be recomputed, even though no logic activity occurred on the output of the gate. If the fault list on the gate output changes, then the fault list must be recomputed forward for gates in the fanout list of that gate, and this must be continued until fault list changes cease. Further complications occur when performing n-value simulation, $n \geq 3$, and when sequential circuit simulation is performed.

3.11 STATISTICAL FAULT ANALYSIS

We have been concerned, up to this point, with modeling faults and performing simulation on circuits in such a way that the effectiveness of a test program is determined by how many of the faults modeled in the circuit are detected. The objective was to (a) get an accurate accounting of how many of the faults are detected and (b) use this as a figure of merit for the test program. If the percentage of faults detected is too low, then more test vectors must be created and fault simulated against the remaining undetected faults. This is repeated iteratively with different sets of test vectors in order to boost the fault coverage to an acceptable level.

The purpose of statistical fault analysis (Stafan) is to obtain an estimate of the fault coverage without simulating all of the faults.[10,11] A logic simulation is performed on the circuit. During the logic simulation, statistics are compiled at the various internal nodes. These statistics involve counting the numbers of 1s and 0s that occur on each internal net. The following entities are defined for each net in the circuit:

$C1(n)$—the *one-controllability*, the probability of net n having a value of one on a randomly selected vector

$C0(n)$—the *zero-controllability*, the probability of net n having a value of zero on a randomly selected vector

$B1(n)$—the probability of sensitizing a path from net n to a primary output, given that the value of the line is one.

$B0(n)$—the probability of sensitizing a path from net n to a primary output, given that the value of the line is zero.

During logic simulation, counters are maintained for each internal net. The zero-count is incremented at the end of each vector when the value on that net is 0, and the one-count is incremented when the value is a 1. After N vectors, the one- and zero-controllabilities are computed as $C1(n)$ = one-count/N and $C0(n)$ = zero-count/N. A third counter is maintained for each net. It is called the sensitization counter. It is incremented if the net is sensitized to the output of the gate that it is driving. For an n-input AND gate, input j is sensitized to the output if all other inputs are at logic 1. For an OR gate, input j is sensitized to the output if all other inputs are at logic 0. After N vectors, the one-level sensitization probability for net n is computed as $S(n)$ = sensitization-count/N.

At the start of simulation, the observabilities of all primary outputs are set to 1. Then, observabilities are computed working back to the inputs. Consider an AND gate with n inputs, and assume the AND gate drives net p. A value of 1 on input j is observable at p only when all inputs to the gate are at logic 1. This is the same as the probability of $C1(p)$. Note that $C1(p)$ is the joint probability that net j equals one and that j is observable at p. The conditional probability that j is observable at p, given that j is a one, is $C1(p)/C1(j)$. This term can then be used to determine the observability of j. The equation is

$$B0(j) = B0(p) \cdot \frac{S(j) - C1(p)}{C0(j)}$$

To this point there has been an implicit assumption that a net drives only one input. That, however, seldom happens in practice. More likely, a net drives two or more gate inputs. If net j drives two gates with output nets p and q and if their paths to the outputs are completely independent, then the observability of j is the probability of the union of $B1(p)$ and $B1(q)$. However, independent paths are also rare. More likely, the paths to the outputs share common logic. To address this issue, the authors propose the following equation:

$$B1(j) = (1 - a) \max_{1 \le k \le m} \left| B1(i_k) \right| + a \bigcup_{k=1}^{m} B1(i_k)$$

In this equation, i_1 through i_k denote the fanout paths for net j. When $\alpha = 1$, $B1(j)$ is observable independently through each of the m fanout branches, hence the observability is the sum of the observabilities of the branches. However, when $\alpha = 0$, then $B1(j)$ is observable through fanout branches that are interdependent by virtue of divergent and reconvergent logic, so $B1(j)$ is at least as observable as the largest of the individual observabilities.

The discussion so far has centered on combinational circuits. Sequential circuits require a more detailed analysis. Where the sequential nature of the circuit results from cross-coupled NAND or NOR latches, the analysis involves conceptually cutting the loop and analyzing it as an iterative array. Loop counters to count occurrences of loop-sensitization states are also used. The interested reader can find details in the original sources. Here we discuss the actual computations of fault coverage, once the various node statistics are generated during simulation. Assume that we wish to detect an SA1 fault on net j. The probability of detection of that fault is $D1(j) = C0(j) \cdot B0(j)$; that is, it is the joint probability of controlling the net to a zero and the probability of observing a zero on that net.

Given that the probability of detecting a given fault on any single vector is x, then the probability $X(N)$ of detecting that fault by a set of N vectors is $X(N) = 1 - (1 - x)^N$; that is, the probability is one minus the probability of not detecting the fault by any of the N vectors. Because the number of vectors is finite, random errors were shown to produce a biased estimate of fault coverage. Hence, the second term on the right-hand side is divided by a correction factor:

$$W(x) = 1 - \frac{N-1}{6}\beta^2 \frac{x}{1-x}$$

In this correction factor, the term β is a constant of proportionality whose value is determined empirically. With this correction factor, the probability of detecting fault x_i in a test program containing N vectors is

$$f_i(N) = 1 - \prod_{m=1}^{N} \frac{(1 - x_{im})}{W(x_{im})}$$

Once the probability of detection is known for a given fault, the cumulative fault coverage for all K faults, for N vectors, can be determined from

$$F(N) = \frac{1}{K} \sum_{i=1}^{K} f_i(N)$$

How effective is Stafan at predicting fault coverage for a set of test vectors? The authors compared results with those obtained from a deterministic fault simulator on a 64-bit ALU with 4376 faults. A set of 155 vectors produced 75.09% estimated fault coverage. They then ranked the faults according to the probability of detection provided by Stafan. Based on a coverage estimate of 75.09%, 3286 faults with highest probability were assumed to be detected, whereas the remaining 1090 faults were assumed undetected. Of the 1090 undetected faults, 1036 were confirmed to be undetected by the deterministic fault simulator. Of the 3286 faults that were assumed to be detected by Stafan, all but 46 were confirmed to be detected by the deterministic fault simulator. In their investigation of the effectiveness of Stafan, the authors report that setting the parameter $\alpha = 1$ (independent paths to the outputs) gave good correlation with deterministic fault simulation. For β, the value $\beta^2/6 = 5.0$ produced a good match with fault simulation. These values of α and β were found to produce good results on other circuits as well.

3.12 FAULT SIMULATION PERFORMANCE

Feature sizes of integrated circuits have shrunk with remarkable regularity over the years, with the result that increasingly larger numbers of transistors are squeezed onto a given area of silicon each year. One result of all this is that fault simulation of large circuits can take many hours, or days. Hence, fault simulation performance is of vital importance. It was pointed out at the beginning of this chapter that growing circuit size implied a growing fault list as well as a larger number of test vectors required to stimulate all the faults in the circuit. These three parameters—circuit size, fault count, and number of vectors—suggest that simulation time may, in the worst case, increase in proportion to the third power of circuit size. As a result, it is vitally necessary to exploit every possible opportunity to improve fault simulation performance.

Consider the performance of parallel fault simulation. A compiled, zero-delay fault simulator is not able to correctly predict the behavior of asynchronous circuits where correct response depends on being able to recognize and process critical propagation delays. It will only handle combinational and synchronous sequential circuits. When fault simulating a synchronous sequential circuit and processing 31 faults in parallel, together with the fault-free circuit, the parallel fault simulator must simulate all of the vectors before processing another 31 faults, unless all of the faults are detected before the end of the vector set is reached. (If a design implements full scan, it can be considered to be a combinational circuit for purposes of analysis.)

The PPSFP fault simulator, by virtue of the fact that it simulates multiple vectors in parallel, is only able to process combinational or full-scan circuits. However, in this restricted environment, it is capable of operating extremely fast. In combinational circuits, it is not uncommon for many (most) faults to be detected in the first 10 to 15 vectors. For these faults it only requires a single pass through the fault simulator to detect the fault and delete it from further consideration because PPSFP is simultaneously simulating 32 vectors.

Dropping faults in the parallel fault simulator is more complicated because 31 faults are processed in parallel, and the vectors are usually simulated until all are detected. The probability of selecting 31 faults that will all be detected before the end of the simulation is usually quite low. It is possible to check the number of faults detected at various points during simulation and, when some threshold is reached, stop simulating that group of faults and restart with a new set, where the undetected faults from the terminated group are kept and undetected faults from the fault list are added to replace the faults that are dropped. That, of course, introduces some redundancy into the process. Parallel fault simulation is one method that would benefit from states applied analysis.

Numerous methods have been devised to speed up fault simulation. Some of them were previously discussed, including fault dropping, states applied analysis, and simulating only one representative fault from a set of equivalent faults. Other methods for improving performance of fault simulation include rank-ordering, rearranging vectors, and statistical fault simulation.

It was mentioned in Section 2.6 that the circuit model for a compiled simulator had to be rank-ordered in order to get correct results. Rank-ordering can also benefit concurrent fault simulation. Given a circuit in which all or most of the circuit elements have zero delay, if the logic elements are simulated in random order, some of the elements may be simulated multiple times during each vector. This is especially true for large combinational blocks. In one particular incident, this author was fault simulating a large combinational array multiplier in which the elements all had zero delay and were randomly positioned in the circuit model. A counter inserted in the fault simulator for debug purposes indicated that some logic gates in the cones of the high-order output bits were being simulated a hundred times or more during each vector. After rank-ordering and resimulating the circuit so that no element was simulated until all its predecessors had been simulated, fault simulation time was reduced from almost a full day down to about an hour.

When a concurrent fault simulator processes a combinational circuit, the amount of activity during fault simulation is affected by the number of input event changes that occur during each vector. Again, in some unpublished experiments performed by this author, vectors were randomly applied to the array multiplier. The same vectors were then reordered so as to reduce the number of input events from one vector to the next, and again they were fault simulated. The rearranged vectors produced significantly less total activity during simulation and, as a result, fault simulation time was considerably less. Where pseudo-random vectors are generated and applied to combinational logic, a cursory examination and rearrangement of the vectors, based on Hamming distance (cf. Chapter 10), can yield a significant payback in reduced simulation time.

Statistical fault sampling is another technique that is effective in reducing simulation time for both concurrent and parallel fault simulation. It provides an estimate of fault coverage, and hence the quality of a test, by simulating a small random sample of the faults. Sufficient faults can be simulated to give an arbitrarily high level of confidence that the fault coverage is within some range of the predicted value. Statistical fault simulation can be preceded by a states applied analysis.[12] If analysis reveals that the percentage of potentially detectable faults is not sufficient to yield the required fault coverage, then there is no point in performing fault simulation until the percentage of potentially detected faults is increased.

It is possible to combine the features of parallel and concurrent fault simulation.[13] The parallel value list (PV) simulates all faults in one pass, as in concurrent fault simulation, but stores faulty values using individual bit positions in a word. Each fault is uniquely identified by a group number and bit position pair. Faults grouped together in a given parallel value word are chosen based on their proximity to one another. If they are close together in the circuit and if no activity is present in that area of the circuit, the fault word is dropped from forward propagation quickly. The evaluation techniques also differ, depending on whether the output activity occurred on the fault-free or the faulted copy of the gate.

Improvements to the concurrent fault simulation algorithm can be achieved through coding techniques. In one example, a simulation program was reprogrammed to take advantage of the computer architecture.[14] Short loops with many

branches, which can be destructive of performance in a pipelined architecture, were modified via *loop unrolling*. A series of operations was recoded to operate on several contiguous arguments. As an example, the following C code increases the total amount of code but reduces the number of jumps that must be performed.

```
for (i = 0; i < 32; i = i + 4) {
  a(i) = b(i) + k;
  a(i + 1) = b(i + 1) + k;
  a(i + 2) = b(i + 2) + k;
  a(i + 3) = b(i + 3) + k;
}
```

Since many programs are characterized by the fact that a high percentage of CPU time is spent in a small part of the program, identifying high usage code (via software profiling tools) and modifying it can sometimes significantly increase overall performance of the program. In the example just cited, rearranging events for optimized processing led to a reported three-to-one performance enhancement while performing gate-level simulation. In contemporary processors with pipelined architectures, techniques to improve performance may depend heavily on the host workstation, and a technique that provides significant improvement on one workstation may provide little or no improvement on another workstation. Cache size in the host computer also has a bearing on performance. Clearly, the bigger the cache, the better the performance. But, for a given cache size, coding techniques that use code currently in cache, rather than fetching code from main memory, can provide significant payback.

A number of approaches to speeding up fault simulation have involved hardware acceleration architectures. The simplest approach is to use an accelerator architected for design verification. Single faults are injected into the circuit model, and response of the faulted model is compared to that of the fault-free model to determine if the fault causes an incorrect response at an output pin. This is basically an adaptation of the serial fault simulation method. Other accelerator approaches have been designed specifically for fault simulation. Hardware accelerators tend to be competitive when first announced; but because of the rapid rate at which standard workstations evolve in performance, software programs running on the workstations gradually catch up and eventually outpace the accelerators in terms of performance. Being an all-software solution, they enjoy a cost advantage as well, since the workstation can serve both as a fault simulation platform and as a general purpose workstation platform, so when not being used for fault simulation they provide a payback by virtue of being used for other applications.

3.13 SUMMARY

Digital electronics is pervasive: These devices appear in every aspect of our lives, and consumers take for granted the presence of electronic devices that perform control functions found in so many of our appliances, entertainment centers, and modes

of transportation. As a result, consumers are less tolerant of failing devices than they once were. This makes it all the more imperative that devices be verified to be fault-free by manufacturers. That, in turn, makes it imperative that manufacturers employ test programs that are very thorough in ferreting out malfunctioning products. Fault simulation is critical to the performance of this task.

Before the emergence of fault simulation, digital designs were tested using functional test programs that attempted to verify the functionality of PCBs. For small designs, using discrete components, it was not too difficult to identify and exercise all "corners" of the design, as well as all combinations of inputs and internal states. If a faulty product reached a customer, it would be analyzed upon return and a test would be developed targeting that defect. As devices became more complex, and more combinations of inputs plus internal states failed to be tested, it became apparent that test programs would have to be evaluated to quantify their effectiveness at separating good product from bad. Fault simulation programs were developed for this purpose.

Several fault simulation algorithms have emerged over the past three decades. In each instance the objective has been to reduce the number of computations and/or memory requirements in order to render the problem tractable. Some differences in approach result from differences in basic assumptions about the circuit being evaluated. When simplifying assumptions are made, it is possible to take advantage of those assumptions to produce a faster product, but one that will not function correctly when those assumptions do not hold. Hence, the user must understand the capabilities and limitations of the tool that he or she chooses to use in order to obtain maximum benefit from it.

But, even before understanding the algorithms, the user must understand that fault coverage is an approximation to the true thoroughness of a test. Its accuracy depends on the fault model chosen. With greater granularity, a greater number of faults are used in a given circuit to estimate the fault coverage, and the fault coverage estimate will be more accurate. However, generating the estimate will be more time-consuming.

The parallel and concurrent fault simulation algorithms have come to dominate the field. Parallel fault simulation and PPSFP are quite powerful for circuits that conform to design guidelines, including synchronous designs. Concurrent fault simulation requires more memory to perform effectively, but it is able to handle a wider range of circuits, synchronous or asynchronous, as well as many more defect modes.

The deductive fault simulator was once used in at least one commercial fault simulator (LASAR—logic automated stimulus and response), but it doesn't have the speed advantage of parallel fault simulation for synchronous circuits and it doesn't have the robustness of concurrent fault simulation for asynchronous circuits. One interesting feature of LASAR was the use of the NAND gate to model all logic elements. It's been well known since early in the twentieth century that NAND gates could be used to model any other logic element.[15] By relying on a single primitive, the processing rules for deductive fault simulation were greatly simplified.

With growing circuit size, increased use of core modules, and the appearance of more memory arrays in circuit models, the need for behavioral simulation capability

is growing. In fact, the ideal fault simulator will be able to process circuits ranging from transistor level to high-level RTL. The concurrent fault simulator fits these requirements; other fault simulation technologies come up short at one end or the other, or both.

Effective use of simulation requires a knowledge of the design environment in which the tools will be used. Assumptions that hold in one design environment may not hold in another. Tools developed for use in combinational or synchronous sequential designs may give totally inaccurate results if applied to asynchronous sequential designs. On the other hand, the synchronous design environment permits simplifying assumptions that can help to speed up simulation. However, performance improvements in some instances are gained at the expense of generality; the algorithms simply will not work on many circuits.

Many claims are made for the various algorithms that have been published over the years. Making comparisons is difficult, because an algorithm that is quite efficient on one circuit may perform rather poorly on other circuits. Some of the performance advantages may be inherent in the algorithms, with a particular algorithm being "tuned" to recognize and apply special processing techniques to certain, commonly occurring circuit configurations. But some of the performance advantages seen in practice may be more a result of a general proficiency with which the algorithms are coded. Effective coding can cause an algorithm to perform as much as two or three times more efficiently than it might otherwise perform. Fault simulation is one of those applications where 5–10% of the software code consumes 95% of the execution time. Recognizing and optimizing that 5–10% of the code can yield a significant payback.

PROBLEMS

3.1 Create the truth table for a three-input OR gate corresponding to that of the AND gate in Figure 3.5. Show the response for SA0 faults on the inputs and the SA0 and SA1 faults on the output.

3.2 Given a four-input AND gate with six faults: SA1 on each of the four inputs, and SA0 and SA1 on the output. Applying the following five vectors toggles all pins to 0 and 1 : $A,B,C,D = \{(1000), (0100), (0010), (0001), (1111)\}$. What is the fault coverage?

3.3 Given a 32-bit ALU with two 32-bit input ports, a carry-in, and five function select bits (i.e., a total of 70 inputs), the test engineer creating the test program decides to simply apply all possible combinations to the inputs. If vectors are applied and response evaluated at the rate of 10,000,000 test vectors per second, how long will it take to exhaustively test the circuit?

3.4 In Section 3.6 it was stated that detection of a fault could not be claimed if the fault-free circuit responds with X and the faulty circuit responds with 0 or 1. Why?

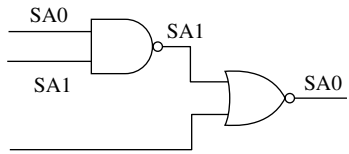


Figure 3.15 Dominance relationships.

3.5 The bufif0 in Figure 3.6 drives a bus. If the enable is not active, the bus is floating (disconnected from the driver). One way to cope with this situation is to connect the bus to a pullup or pulldown. Then, if no driver is actively driving the bus, the bus assumes a weak 1 (H) or a weak 0 (L) value that can be overcome by an active 1 or 0. Recreate the truth table in Figure 3.6, assume the existence of a pullup, and replace the Z's by H's. Explain how to detect the stuck-at faults F_1 through F_5 in this situation.

3.6 A commercial fault simulator is likely to create 12 faults for the multiplexer in circuit in Figure 3.7; identify them.

3.7 Generate a list of stuck-at faults for each of the primitive logic gates in Figure 2.44. Using dominance and equivalence properties, collapse the fault lists.

3.8 Given the following sets Ta through Te of tests for faults a, b, c, d, e, show all dominance and equivalence relationships between these test sets.

Ta = {t1, t2, t3, t4, t5}

Tb = {t3, t4}

Tc = {t3, t4, t6, t7}

Td = {t3, t4}

Te = {t2, t8}

3.9 Identify the dominance and equivalence relationships between the four faults in the circuit of Figure 3.15.

3.10 Prove the dominance and equivalence theorems.

3.11 The circuit on the left, in Figure 3.16, is represented on the right by a functional block. Find a set of vectors that detect all SA0 and SA1 faults on the pins of the functional block model but fails to detect a SA1 on the top input to AND gate *D* in the gate-level model.

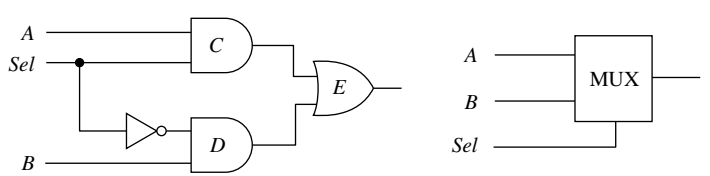


Figure 3.16 Hidden fault.

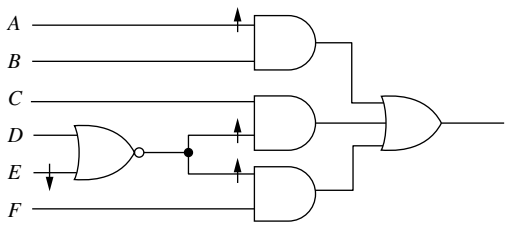


Figure 3.17 Using deductive fault simulation.

3.12 Finish the fault simulation example for Figure 3.10 in Section 3.6.1. What is the result vector at the outputs of AND gate *J* and XOR *K*?

3.13 In the circuit of Figure 3.10, assume 10 faults: SA1 faults on the inputs to gates *G*, *H*, and *J*, SA0 faults on the inputs to gate I, and an SA1 fault at input *E*. The following four vectors are applied to the circuit: *A,B,C,D,E,F* = {(000011), (010110), (110001), (001101)}. Perform parallel fault simulation on the circuit and identify the faults detected by each vector. Perform states applied analysis; is there any savings in computation time?

3.14 Perform parallel pattern single fault propagation (PPSFP) on the circuit of Figure 3.10 using the faults and vectors defined in the preceding problem.

3.15 Again using the circuit in Figure 3.10, and the faults and vectors defined in problem 3.13, use Stafan to estimate fault coverage for the 10 faults.

3.16 The four vectors of Problem 3.13 are applied to the circuit in Figure 3.10, and the fourth vector responds incorrectly. What faults are most likely to have occurred? What faults are most likely not to have occurred?

3.17 The circuit in Figure 3.17 has four stuck-at faults, indicated by the arrows. Two vectors are applied: *A,B,C,D,E,F* = {(011011), (011111)}. Use deductive fault simulation to determine all of the faults detected by each of the two vectors.

3.18 Using concurrent fault simulation, along with the four faults and two input vectors from the previous problem, determine which of the four are detected. Show your work.

3.19 Using PPSFP, find all input combinations that will detect a SA0 fault on the input to gate *I* that is driven by gate *H* in Figure 3.10. Find all combinations that will detect a SA1 on the lower input to gate *K*.

3.20 It was stated in Section 2.7 that a circuit had to be rank-ordered in order to get correct results with a compiled simulator. Is that strictly correct? Explain.

3.21 For the circuit in Figure 3.10, write the code for a parallel fault simulator that fault simulates a multiple fault consisting of a SA0 on the output of *G* and a SA1 on the input of *J* driven by primary input *E*.

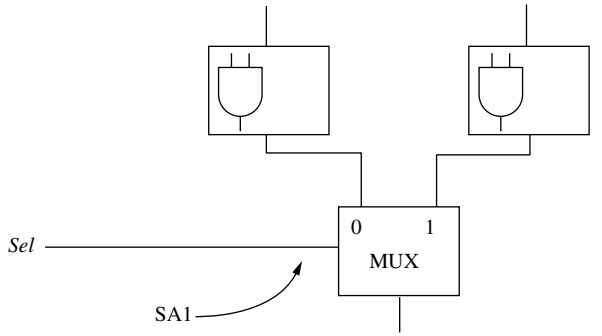


Figure 3.18 A MUX with stuck-at faults.

3.22 For the circuit in Figure 3.10, write the code for a parallel fault simulator that fault simulates a short between the output of *G* and the input of *J* driven by primary input *F*. Assume that the short acts like a wired AND, that is, if either the output of *G* or input *J* is at 0, the entire shorted network assumes the value 0.

3.23 Given the circuit in Figure 3.18, assume three faults: a SA1 on the left input to each of the two indicated AND gates, and a SA1 on the select line *Sel*. Which of the three faults can be detected when *Sel* is set to 0?

3.24 Joe bought a very old house and had Sam the electrician rewire the light switches in the stairwell leading to the upstairs bedrooms so that the light could be turned on and off both at the foot of the stairs and at the upstairs landing. When Sam completed the wiring he turned on the circuit breaker and the light came on. He went upstairs and flicked the switch to both positions, and the light went off and came back on. Sam went downstairs and repeated the exercise, with successful results. He then turned the light off. Later that night Joe awakened and decided to go downstairs and check out the refrigerator. He flipped the light switch but the light did not turn on. Explain what happened.

REFERENCES

1. Eldred, R. D., Test Routines Based on Symbolic Logic Statements, *J. ACM*, Vol. 6, No. 1, January 1959, pp. 33–36.
2. Wadsack, R. L., Fault Coverage in Digital Integrated Circuits, *Bell Syst. Tech. J.*, May–June 1978, pp. 1475–1488.
3. Mei, K. C. Y, Fault Dominance in Combinational Circuits, *Digital Syst. Lab., Report No. 2*, Stanford University, August 1970.
4. Case, G. R., SALOGS-IV, A Program to Perform Logic Simulation and Fault Diagnosis, *Proc. 15th D.A. Conf.,* 1978, pp. 392–397.
5. Waicukauski, J. A. et al., Fault Simulation for Structured VLSI, *VLSI Syst. Des.*, Vol. 6, No. 12, December 1985, pp. 20–32.

6. Ulrich, E. G., and T. Baker, Concurrent Simulation of Nearly Identical Digital Networks, *Computer*, Vol. 7, No. 4, April 1974, pp. 39–44.
7. Schuler, D. M., and R. K. Cleghorn, An Efficient Method of Fault Simulation for Digital Circuits Modeled from Boolean Gates and Memories, *Proc. 14th D.A. Conf.*, 1977, pp. 230–238.
8. Cheng, W., and M. Yu, Differential Fault Simulation for Sequential Circuits, *J. Electron. Testing: Theory and Applications*, Vol. I, 1990, pp. 7–13.
9. Armstrong, D.B., A Deductive Method for Simulating Faults in Logic Circuits, *IEEE Trans. Comput.*, Vol. C-21, No. 5, May 1972, pp. 464–471.
10. Jain, S. K., and V. D. Agrawal, Statistical Fault Analysis, *IEEE Des. Test*, Vol. 2, No. 1, February 1985, pp. 38–44.
11. Jain, S. K., and V. D. Agrawal, STAFAN: An Alternative to Fault Simulation, *Proc. 21st D.A. Conf.*, 1984, pp. 18–23.
12. Case, G. R., A Statistical Method for Test Sequence Generation, *Proc. 12th D.A. Conf.,* 1975, pp. 257–260.
13. Moorby, P. R., Fault Simulation Using Parallel Value Lists, *Proc. ICCAD,* 1983, pp. 101–102.
14. Krohn, H. E., Vector Coding Techniques for High Speed Digital Simulation, *Proc. 18th D.A. Conf.,* 1981, pp. 525–529.
15. Sheffer, H. M., A Set of Five Independent Postulates for Boolean Algebras, *Trans. Am. Math. Soc.*, Vol. 14, 1913, pp. 481–488.

CHAPTER 4

Automatic Test Pattern Generation

4.1 INTRODUCTION

In Chapter 3 we looked at fault simulation. Its purpose is to evaluate test programs in order to measure their effectiveness at distinguishing between faulty and fault-free circuits. The question of the origin of test stimuli was ignored for the moment; we simply noted that test programs could be derived from test stimuli originally intended for design verification, or stimuli could be written specifically for the purpose of exercising the circuit to reveal the presence of physical defects, or stimuli could be produced by an automatic test pattern generator (ATPG). We now turn our attention to the ATPG. However, we also examine two alternatives to fault simulation in this chapter: testdetect and critical path tracing. These two methods share much common terminology, as well as methodology, with corresponding ATPGs, so it is convenient to group them with their corresponding ATPGs.

A number of techniques have emerged over the past three decades to generate test programs for digital circuits. For combinational circuits several of these, including D-algorithm, PODEM, FAN and Boolean differences, have been shown to be true algorithms, in the sense that, given enough time, they will eventually come to a halt; that is, there is a stopping rule. If one or more tests exist for a given fault, they will identify the test(s). For sequential circuits, as we will see in the next chapter, no such statement can be made. Push-button solutions capable of automatically generating comprehensive test programs for sequential circuits require assistance in the form of design-for-test (DFT), which will be a subject for a later chapter. In this chapter, we will examine the algorithms and procedures for combinational logic and attempt to understand their strengths and weaknesses.

4.2 THE SENSITIZED PATH

In Section 3.4, while discussing the stuck-at fault model, it was pointed out that whenever fault modeling alternatives were considered, combinatorial explosion

Digital Logic Testing and Simulation, *Second Edition*, by Alexander Miczo
ISBN 0-471-43995-9 Copyright © 2003 John Wiley & Sons, Inc.

resulted. The number of choices to make, or the number of problems to solve, literally explodes. The stuck-at fault model is a necessary consequence of the combinatorial explosion problem. A further consequence of this problem is the *single-fault assumption*. When attempting to create a test, it is assumed that a single fault exists. Experience with the stuck-at fault model and the single-fault assumption indicates that they are effective; that is, a good stuck-at test that detects all or nearly all single stuck-at faults in a circuit will also detect all or nearly all multiple stuck-at faults and short faults.

The stuck-at fault has been defined as the fault model of interest for basic logic gates, and tests for detecting stuck-at faults on these gates have been defined. However, individual logic gates do not occur in practice. Rather, they are interconnected with many thousands of other similar gates to form complex circuits. When embedded in a much larger circuit, there is no immediate access to the gate. Hence it becomes necessary to use surrounding circuitry to set up the inputs to the gate under test and to cause the effects of the fault to travel forward and become visible at an output pin where these effects can be observed by a tester.

4.2.1 The Sensitized Path: An Example

The circuit in Figure 2.43, repeated here as Figure 4.1, will be used to illustrate the process. The goal is to find a test for an SA0 on input 3 of gate *K* (i.e., the input driven by gate *H*; on schematic drawings, inputs will be numbered from top to bottom). Since gate *K* is an OR gate, the test for input 3 SA0 requires that input 3 be set to 1 and the other inputs be set to 0. Two problems must be solved: First, logic values must be computed on the primary inputs that cause the assigned test values to appear at the inputs of *K*. Second, the values assigned to the primary inputs must make the fault effect visible at the output. In addition, the values computed on the primary inputs during these operations must not conflict.

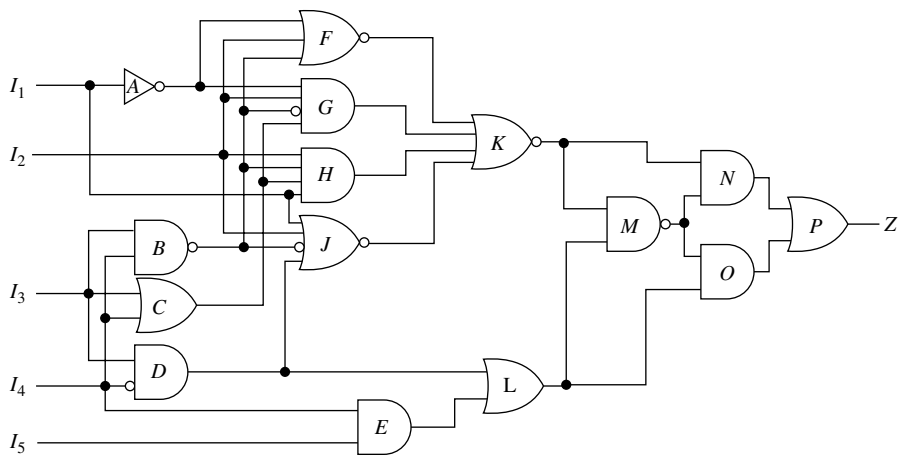


Figure 4.1 Sensitizing a path.

We attempt to create a sensitized path from the fault origin to the output. A *sensitized path* of a fault f is a signal path originating at the fault origin f whose value all along the path is functionally dependent on the presence or absence of the fault. If the sensitized path terminates at a net that is observable by test equipment, then the fault is *detectable*. From the response at the output, it can be determined whether or not the targeted fault occurred. The process of extending a sensitized path is called *propagation*.

Gate H, which drives the faulted input of gate K, is an AND gate, and a logic 1 on its output only occurs if all its inputs have logic 1 values. This is called *implication*; a 1 on the output of an AND gate implies logic 1 on all its inputs. This implication operation can be taken a step further. The top input of H is driven directly by I_2, and its bottom input is driven by I_1. Hence, both of these inputs must be assigned a logic 1. This implication operation can be applied yet again. A 1 on the input to inverter A implies a 0 on its output, and that 0 drives gate G. Therefore, the output of gate G is a 0. Fortunately, that 0 is consistent with the initial values assigned to the inputs of K. Other implications remain. I_2 drives NOR gate F with a 1, causing the output of gate F to become 0. Again, that value is consistent with the original assignments to K. Finally, I_1 drives NOR gate J, and gate J responds with a 0, so once again the assignment is consistent with the required values on K.

All that remains to get a 1 from gate H is to get 1s from gate B and gate C. Gate B is a two-input NAND gate, and it generates a 1 if either of its inputs is a 0. We choose I_3 and set it to 0. We still need to get a 1 from gate C. It is a two-input OR gate and its upper input, from I_3, was already set to 0. So, we set I_4 to 1.

All of the inputs to K have now been satisfied, so the output of K is a 0 if the NOR gate is operating correctly, and the output of K is 1 if the fault exists. At this point we introduce the D-notation. The letter D (discrepancy) represents a *composite signal* 1/0, where the first number represents the value on the fault-free circuit, and the second number represents the value on the faulty circuit. The letter $\overline{\text{D}}$ represents the composite signal 0/1, meaning that the fault-free circuit has the value 0 and the faulty circuit has the value 1. The output of gate K is $\overline{\text{D}}$.

A $\overline{\text{D}}$ will now be propagated forward through gate M. To do so requires a logic 1 on the other input to M, driven by gate L. The output of gate D is a 0, by virtue of the 0 on input I_3. However, a 1 can be obtained from gate E by assigning a 1 to input I_5. All of the inputs have now been assigned; the values are $I_1,I_2,I_3,I_4,I_5 = (1,1,0,1,1)$.

However, a problem seems to appear. NAND Gate M has a $\overline{\text{D}}$ and a 1 on its inputs. That produces a D on the output. Now, gate N has a $\overline{\text{D}}$ and a D on its inputs. That means that the fault-free circuit applies 0 and 1 to gate N, and the faulty circuit applies 1 and 0. So both the fault-free and the faulty circuits respond with a 0 on the output of gate N. One solution is to back up to the last assignment, $I_5 = 1$, and change it to $I_5 = 0$, so that the assignments on the primary inputs are I_1, I_2, I_3, I_4, $I_5 = (1,1,0,1,0)$. Then, the output of E becomes 0. That causes the output of L to become 0, which in turn causes the output of M to become 1. A $\overline{\text{D}}$ and 1 on the input to N cause a $\overline{\text{D}}$ to appear on its output. Since $L = 0$, the other input to P is 0, and the $\overline{\text{D}}$ makes it through P to the output Z. As we will see, if we had considered all possible propagation paths, this last operation, changing the value on I_5, would not have been necessary.

4.2.2 Analysis of the Sensitized Path Method

The operation that just took place will now be analyzed, and some observations will be made. The process of backing up and changing assignments is called *justification*, also sometimes referred to as the *consistency* operation. The two processes, propagation and justification, can be used to find a test for almost any fault in the circuit (redundant logic, as we shall eventually see, presents testing problems). Furthermore, propagation and justification can be applied in either order. We chose to start by propagating from the point of fault to an output. It would be possible to first justify the assignments on the four inputs of gate *H*, then propagate forward to the output, one gate at a time, each time justifying all assignments made in that step of the propagation.

During the propagation phase all required assignments are placed on the assignment stack. Then, in the justification phase, the assignment stack expands and contracts. When the stack is finally empty, the justification phase is complete. In the second approach, processing begins with the justification process, attempting to satisfy initial assignments on the gate whose input or output is being tested. Each time the assignment stack empties, control reverts to the propagation mode and the sensitized path extends one gate closer to the outputs. Then, control again reverts to the justification routine until the assignment table is again empty. Control passes back and forth in this fashion until the sensitized path reaches an output and all assignments are satisfied.

Implication When assignments are made to individual gates, they sometimes carry implications beyond the immediate assignment. An *implication* is an assignment that is a direct consequence of another assignment. Only one assignment is possible. Consider the assignment of a logic 1 to the output of gate *H*. This implied that all of its inputs must be 1, implying that I_1 and I_2 must both be 1. Once I_1 had been assigned a 1, that implied a 0 on the output of inverter *A*, which in turn implied a 0 on the output of *G*. These operations will be stated more formally in a later section, because now it is sufficient to point out that these implications obviated the need to make choices at various points during the operation.

The Decision Table During propagation and justification, gates are encountered where choices must be made. For example, when a 0 was required from the NOR gate labeled *F*, the value 1 was assigned to the upper input. This choice caused a problem because it resulted in an assignment $I_1 = 0$ that conflicted with a previous assignment $I_1 = 1$. Because a choice existed, it was possible to back up and make an alternate choice that eventually proved successful. In large, complex circuits with much fanout, complex multilevel decisions often must be made. If all decisions at a given gate have been tried without success, then the decision stack must be popped and a decision made at the next available decision point. Furthermore, assignments to all gates following the point at which the decision was made must be erased, and any mechanism used to keep track of decisions for the gate that was popped off the decision stack must be reset. The *decision table* maintains a record of choices, or alternatives.

The implication operation is of value here because it can often eliminate a number of decisions. For example, the initial test for gate *H* assigned a logic 1 to input I_2. But assigning a 1 to I_2 forces—that is, implies—a 0 on the output of gate *F*. As a result, if implication is performed, there is no need to justify $F = 0$, and that in turn eliminates the need to make a decision at gate *F*.

The Fault List The fault, input 3 of gate *K*, was selected arbitrarily in order to demonstrate propagation and justification techniques. In actual practice the entire set of stuck-at faults would be compiled into a fault list. That list would then be collapsed using dominance and equivalence (cf. Section 3.4.5). Each time a test vector is created for a fault in the circuit, that test vector would be fault simulated in order to determine if any other faults are detected. The objective is to avoid performing test vector generation on faults that have already been detected.

For example, the test for input 3 of *K* SA1 causes the fault-free circuit to assume the value $Z = 0$. If input 3 of *K* were actually SA1, the output would assume the value 1. But several other faults would also cause *Z* to assume the value 1, the most obvious being the output of *P* SA1. Other faults causing a 1 output include outputs of gate *N* or gate *O* SA1. In fact, any fault along the sensitized path that causes the value on that path to assume a value other than the correct value will be detected by the test vector.

The importance of this observation lies in the fact that if we can determine which previously undetected faults are detected by each new test vector, then we can check them off in the fault list and do not need to develop test vectors to specifically test for these faults. Several techniques for accomplishing this will be described later.

Making Choices The sensitized path method for generating tests was used during the early 1960s.[1] When this method reached a net with fanout during propagation, it arbitrarily selected a single path and continued to pursue its objective of reaching an output. Unfortunately, this blind pursuit of an output occasionally ignored easy solutions.

Consider what happens when an attempt is made to propagate a test through gate *M* in Figure 4.2. Assume that the inputs to gates *M* and *Q* are primary inputs and that the upper input to gate *N* is driven by other complex logic. Assume also that gate *P* drives a primary output while gate *N* drives other complex logic. Gate *P* is not difficult to control. Its lower input, driven by gate *Q*, can be set to 1 with a 0 at either input to *Q*. Gate *N* represents greater difficulties because a logic assignment at its upper input must be justified through other logic, and a test at its output must be propagated through additional logic. An arbitrary propagation choice could result in an attempt to drive a test through the upper gate. In fact, if a program did not examine the function associated with the fanout to gate *P*, it might go right past a primary output and attempt to propagate a test through complex sequential logic at the output of gate *N*.

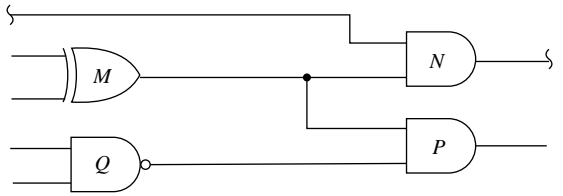


Figure 4.2 Choosing the best path.

By ordering the inputs and fanout list for each gate, the program can be forced to favor (a) inputs that are easiest to control and (b) the propagation path that reaches a primary output with least difficulty whenever a decision must be made. An algorithm called SCOAP, which methodically computes this ordering for all gates in a circuit, will be described in Section 8.3.1.

The Reconvergent Path A difficulty inherent in the sensitized path is the fact that it might not be able to create a test for a fault when a test does exist.[2] This can be illustrated by means of the circuit in Figure 4.3. Consider the output of NOR gate *B* SA0. Inputs I_2 and I_3 must be 0 in order to get a 1 on the output of *B* in the fault-free circuit. In order for the fault to propagate through gate *E*, input I_1 must be 0. Hence the output of *E* is 0 for the fault-free circuit, and it is 1 for the faulty circuit. In order for *E* to be the controlling input to gate *H*, the other three inputs to *H* must be set to 0.

To get a 0 at the output of *F*, one of its inputs must be set to 1. Since the output of *B* is SA0, input I_4 must be set to 1. The output of gate *C* then assumes the value 0 which, together with the 0 on I_3, causes the output of gate *G* to become 1. The sensitized path is now inhibited, so there does not appear to be a test for the fault. But a test does exist. The input assignment (0,0,0,0) will detect a SA0 fault at the output of gate *B*.

4.3 THE D-ALGORITHM

The inability to generate a test for the fault at the output of gate *B* in Figure 4.3 occurred because the sensitized path procedure always attempts to propagate fault

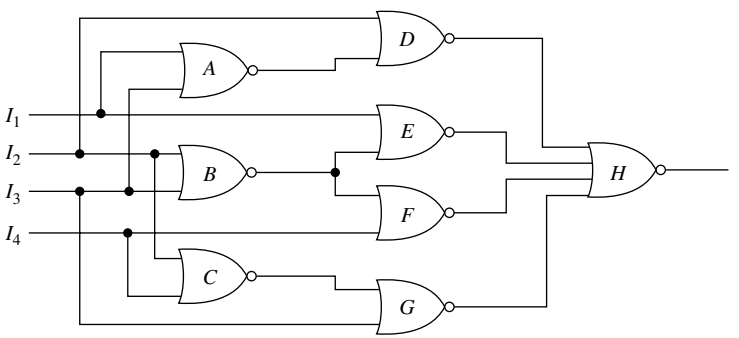


Figure 4.3 Effect of reconvergent fanout.

symptoms through a single path. In the example it was necessary to make a choice because of the presence of fanout. In fact, that was the problem with the first example, that used Figure 4.1. It was not necessary to perform that last operation in which I_5 was changed from 1 to 0. Even though the D and $\overline{D}$ canceled each other out at gate *N*, the D at the output of gate *M* would have propagated through gate *O* and made it to the output as a D. Rather than make a choice, the D-algorithm is capable of propagating a sensitized signal through all paths when it encounters a net with fanout.

We start by formally defining the D-notation of Roth by means of the following table.[3] The D simultaneously represents the signal value on the good circuit (GC) and the faulted circuit (FC) according to the following table:

GC \ FC	0	1
0	0	$\overline{D}$
1	D	1

Conceptually, the D represents logic values on two superimposed circuits. When the good circuit and the faulted circuit have the same value, the composite circuit value will be 0 or 1. When they have different values, the composite circuit value will be D, indicating a 1 on the good circuit and 0 on the faulted circuit, or $\overline{D}$, indicating a 0 on the good circuit and 1 on the faulted circuit.

At the output of gate *B* in Figure 4.3, where a SA0 fault was assigned, the fault-free circuit must have logic value 1; therefore a D is assigned to that net. The goal is to propagate this D to a primary output. Since the output of *B* drives two NOR gates, the D is assigned to an input of gate *E* and to an input of gate *F*. Suppose we require that the other input to both of these NOR gates be the nonblocking value; that is, we assign $I_1 = I_4 = 0$. What value appears at the outputs of *E* and *F*? The inputs are 0 and D on both NOR gates, and the D represents 1 on the good circuit and 0 on the faulted circuit. So NOR gate inputs 0 and 1 are ORed together and inverted to give a 0 on the output of the fault-free circuit, and NOR gate inputs 0 and 0 are ORed and inverted to give a 1 on the output of the faulty circuit. Hence, the outputs of gates *E* and *F* are both $\overline{D}$.

Two sensitized paths, both of which have the value $\overline{D}$, are now converging on *H*. If NOR gates *D* and *G* both have output 0, then the inputs to *H* are (0,0,0,0) for the good circuit and (0,1,1,0) for the faulted circuit. Since *H* is a NOR gate, its output is 1 for the good circuit and 0 for the faulted circuit; that is, its output is a D. However, we are not yet done. We need to obtain 0 from gates *D* and *G*. Since all of the inputs are assigned, all we can do is inspect the circuit and hope that the input assignments satisfy the requirement $D = G = 0$. Luckily, that turns out to be the case.

4.3.1 The D-Algorithm: An Analysis

A small example was analyzed rather quickly, and it was possible to deduce with little difficulty what needed to be done at each step. A more rigorous framework will

now be provided. We begin with a brief description of the cube theory that Roth used to describe the D-algorithm.

A *singular cube* of a function is defined as an assignment

$$(x_1, \ldots, x_n, y_1, \ldots, y_m) = (e_1, e_2, \ldots, e_{m+n})$$

where the x_i are inputs, the y_j are outputs, and $e_i \in \{0, 1, X\}$. A singular cube in which all input coordinates are 0 or 1 is called a *vertex*. A vertex can be obtained from a singular cube by converting all Xs on input coordinates to 0s and 1s.

A singular cube *a contains* the singular cube *b* if *b* can be obtained from *a* by changing some of the Xs in *a* to 1s and 0s. Alternatively, *a* contains *b* if it contains all of the vertices of *b*. The *intersection* of two singular cubes is the smallest singular cube containing all of their common vertices. It is obtained through use of the intersection operator that operates on corresponding coordinates of two singular cubes according to the following table:

I	0	1	X
0	0	—	0
1	—	1	1
X	0	1	X

The dash (—) denotes a *conflict*. If one singular cube has a 0 in a given position and the other has a 1, then they are in conflict; the intersection does not exist. Two singular cubes are *consistent* if a conflict at their output intersections implies a conflict on their input intersections. In terms of digital logic, this simply says that a stimulus applied to a combinational logic circuit cannot produce both a 1 and a 0 on an output. The term *singular* is used to denote the fact that there is a one-to-one mapping between input and output parts of the cube. We will henceforth drop the term singular; it will be understood that we are talking about singular cubes. Furthermore, to simplify notation, we will restrict our attention in what follows to single output cubes, the definitions being easily generalized to the multiple output case.

A *cover C* is a set of pairwise consistent, nondegenerate cubes, all referring to the same input and output variables. Given a function *F*, a *cover of F* is a cover *C* such that each vertex $v \in F$ is contained in some $c \in C$. A *prime* cube of a cover is one that is not contained in any other $c \in C$. If the output part of a cube has the value 0, the cube will be called a 0-point; if it has value 1, it will be called a 1-point; and if it has value X (don't care), it will be called an X-point. An *extremal* is a prime cube that covers a 0-point or 1-point that no other prime cube covers.

Example The function $F = a_0a_1 + \bar{a}_0a_2$ can be represented by the cube of Figure 4.4. The set of vertices for this cube is as follows:

a_0	a_1	a_2	F
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	1

■■

The following is a covering for the function which consists of prime cubes (asterisks denote extremals):

*	1	1	X	1	
	X	1	1	1	p_1
*	0	X	1	1	
*	1	0	X	0	
	X	0	0	0	p_0
*	0	X	0	0	

The set of cubes for which the output is a 1 is denoted p_1. Likewise, p_0 denotes the set of cubes whose output is 0. The reader should verify that each vertex of F is contained in at least one extremal. Two intersections follow:

X	1	1	1		1	0	X	1
0	X	1	1		0	X	0	0
0	1	1	1			—		

In the first intersection the cube (0, 1, 1, 1) is the smallest cube that contains all points common to the two vectors intersected. The second intersection is null. From Figure 4.4 it can be seen that the two cubes have no points in common. The set of extremals contains all of the vertices; hence it completely specifies the function for all defined outputs.

The reader familiar with the terms "implicant" and "prime implicant" may note a similarity between them and the cubes and extremals of cube theory. An *implicant* is a product term that covers at least one 1-point of a function F and does not cover any 0-points. An implicant is *prime* if

1. For any other implicant there exists a 1-point covered by the first implicant that is not covered by the second implicant, and
2. When any literal is deleted, the resulting product term is no longer an implicant of the function.

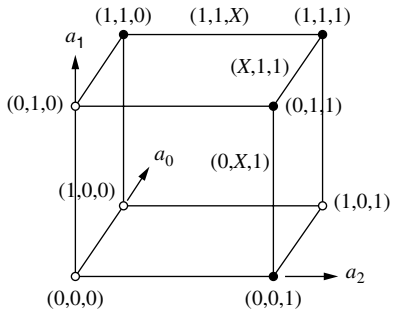


Figure 4.4 Cube representation of a function.

Implicants and prime implicants deal with product terms that cover 1-points, whereas cubes deal with both 1-points and 0-points. The cover corresponds to the set of implicants for both the function F and its complement $\overline{F}$. The collection of extremals corresponds to the set of prime implicants for both the function F and its complement $\overline{F}$.

4.3.2 The Primitive D-Cubes of Failure

A *primitive* is an element that cannot be further subdivided; processing power is built into the D-algorithm. Up to this point the basic switching gates have been regarded as primitives. As we shall see, the D-algorithm can accommodate primitives that are composites of several basic switching gates. A fault model for the D-algorithm is called a primitive D-cube of failure (PDCF). The two-input AND gate will be used to describe the procedure for generating a PDCF. We start with a cover for the AND gate, in which the input vertices are numbered 1 and 2, and the output vertex is number 3.

1	2	3	
0	0	0	p_0
0	1	0	
1	0	0	
1	1	1	p_1

If input 1 is SA1, then the output is completely dependent on input 2. The cover then becomes

1	2	3	
0	0	0	f_0
1	0	0	
0	1	1	f_1
1	1	1	

(When referring to the faulted circuit, the set of 0-points is denoted as f_0 while the set of 1-points is as f_1.) We now have two distinct circuits. The first one produces an output of 1 only when both inputs are at 1. The second circuit produces an output of 1 whenever the second input is a 1, regardless of the value applied to the first input. A cursory examination of the two sets of vertices reveals an input combination, (0,1), that causes a 0 output from the fault-free circuit and a 1 from the faulted circuit. The vector (0,1) is clearly, then, a test for the presence of the SA1 fault on input 1.

Are there any other tests for input 1 SA1? The answer can be determined by performing a point-by-point comparison of vertices from the two sets of vertices. In this case, there is only one test for input 1 SA1. This test is the PDCF for the SA1 fault on input 1 of the AND primitive. The comparison of vertices from the two sets can be performed using the intersection table of the previous section. When we get to the output, we do not flag it as a conflict; rather, we assign a $\overline{D}$, where D and $\overline{D}$ have the meanings described previously.

If the two-input AND gate is faulted with its output SA1, the cover for this faulted two-input AND gate becomes

1	2	3	
0	0	1	f_1
0	1	1	
1	0	1	
1	1	1	

There are three tests for the output SA1, and any of these tests can be chosen for the fault. However, from the first two entries it is observed that the second input can be either a 0 or a 1 (i.e., its value does not matter), suggesting the test (0, X). Likewise, from the first and third entries it can be concluded that (X, 0) is a test for the fault. The value of this observation lies in the fact that only one input needs to be assigned. Can this be computed algorithmically?

Consider again the input SA1 fault for the two-input AND gate. The cover for the good circuit can be described in terms of extremals. For the good circuit the cover is

1	2	3	
0	X	0	p_0
X	0	0	
1	1	1	p_1

For the faulted gate the cover is

1	2	3	
X	0	0	f_0
X	1	1	f_1

The vertex (0,1) is contained in the input parts of the cubes (0, X, 0) $\in p_0$ and (X, 1,1) $\in f_1$. The input parts of these two cubes can be intersected to yield the original vertex (0,1). The intersection of an element from p_0 with an element from f_1 has produced a test for input 1 of the AND gate SA1. This, then, suggests the following general method for finding test(s) for a particular fault:

1. Create a cover consisting of extremals for both the fault-free and faulted circuits.
2. Intersect members of f_0 with members of p_1.
3. Intersect members of f_1 with members of p_0.

Since there must be at least one vertex that produces different outputs for the good circuit and faulted circuit (why?), either step 2 or step 3 (or both) must result in a non-empty intersection. Note that the intersections need not necessarily result in a vertex.

Example Consider the output of the two-input AND gate SA1. The cover f_1 consists of the single cube (X, X, 1). Intersecting it with the extremals in p_0 results in the two tests (0, X, $\overline{\text{D}}$) and (X, 0, $\overline{\text{D}}$). (When performing steps 2 and 3 above, only the input parts are intersected.) ■■

PDCFs were developed for a rather elementary circuit, namely an AND gate. We leave it as an exercise for the reader to develop PDCFs for other elementary gates such as OR, NAND, NOR, and Invert. We point out that the technique for creating PDCFs is quite general. Given a cover for a circuit G and its faulted counterpart, the method just described can create a test for the circuit. As an example, consider the AND-OR-Invert (AOI) of Figure 4.5. The circuit with input 1 SA1 is denoted G^*. The Karnaugh maps for G and G^* are

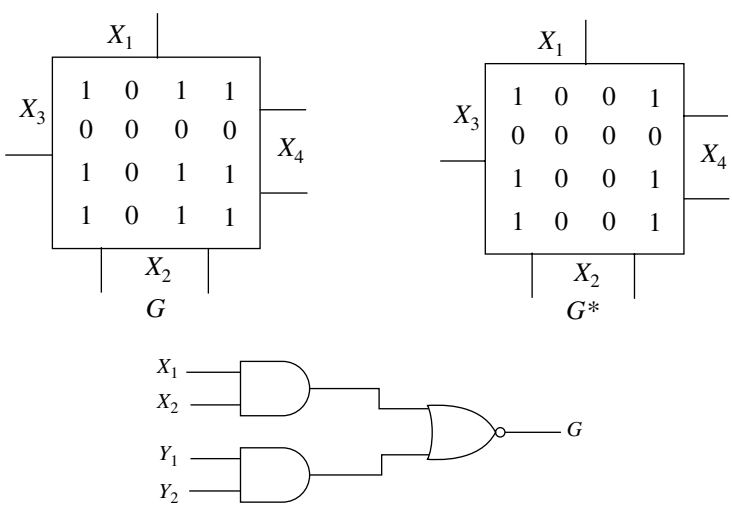


Figure 4.5 AND-OR-Invert (AOI) circuit.

The extremals for G and G^* are

X_1	X_2	X_3	X_4	G		
1	1	X	X	0	}	p_0
X	X	1	1	0		
0	X	0	X	1	}	p_1
0	X	X	0	1		
X	0	0	X	1		
X	0	X	0	1		

X_1	X_2	X_3	X_4	G^*		
X	1	X	X	0	}	f_0
X	X	1	1	0		
X	0	0	X	1	}	f_1
X	0	X	0	1		

The complete set of intersections $p_0 \cap f_1$ and $p_1 \cap f_0$ yields

0	1	0	X	D
0	1	X	0	D

Either of these two vectors will distinguish between the fault-free circuit and the circuit with input 1 SA1.

4.3.3 Propagation D-Cubes

The D-algorithm provides methods for processing circuits composed of a network of primitives. Associated with each primitive is a set of rules for propagating tests through it and for justifying test assignments from its outputs back to its inputs. During propagation a sensitized signal, D or $\overline{\text{D}}$, appears at one or more inputs to a primitive, and the remaining inputs must be assigned logic values that cause the output to be totally dependent on the sensitized signal. It is also assumed, in keeping with the single-fault assumption, that the primitive through which the fault is propagating is fault-free; that is, the fault of interest occurred elsewhere and the task is to drive it to an observable output.

Since the goal is to drive a test through the primitive, a situation must be created in which the response at the output of the primitive in the fault-free circuit is 1 and the response at the output of the primitive in the faulted circuit is 0, or conversely. This tells us that if the input part of the cube for the primitive in the fault-free circuit is in p_0, then the input part of the cube for the primitive in the faulted circuit must be in p_1, and vice versa. This suggests that we again want to perform intersections. We will perform intersections, but the previous intersection table cannot be used

because it prohibited conflicts. We are now actually looking for conflicts so we use the following table:

	0	1	X
0	0	$\overline{\text{D}}$	0
1	D	1	1
X	0	1	X

The row and column labels represent the values on input i of the first and second cubes, respectively. Since elements from p_0 are intersected with elements from p_1, a conflict will always appear on the output. A conflict will also appear on at least one input coordinate position. If all possible intersections are performed, a table of entries called *propagation D-cubes* is created. Then, when a signal must propagate through a primitive, a search is made through the table for an entry with D and $\overline{\text{D}}$ values that match the signals on the input position(s) of the primitive through which a signal is being propagated. That entry identifies the values that must occur on other inputs to the circuit.

Example Using the cover for the AND-OR-Invert of Figure 4.5, and intersecting p_0 with p_1, the following propagation D-cubes are obtained for the AND-OR-Invert:

1	2	3	4	G
D	1	0	X	$\overline{\text{D}}$
1	D	0	X	$\overline{\text{D}}$
D	1	X	0	$\overline{\text{D}}$
1	D	X	0	$\overline{\text{D}}$
0	X	D	1	$\overline{\text{D}}$
0	X	1	D	$\overline{\text{D}}$
X	0	D	1	$\overline{\text{D}}$
X	0	1	D	$\overline{\text{D}}$

■■

There are actually 16 propagation D-cubes. The other eight are obtained by intersecting p_1 with p_0. They can also be obtained by exchanging D and $\overline{\text{D}}$ signals on both the inputs and outputs. In actual practice it is often necessary to restrict the propagation D-cube tables to contain only those propagation D-cubes having a single D or $\overline{\text{D}}$ among the inputs. That is because it is possible to have as many as 2^{2n-1} propagation D-cubes for a function with n inputs. For a function with 6 inputs, this could result in a table of 2048 entries if all single and multiple D and $\overline{\text{D}}$ signals were maintained on the inputs. Multiple D and $\overline{\text{D}}$ values on the inputs are needed much less frequently than single D or $\overline{\text{D}}$ signals and can be created from the cover when needed.

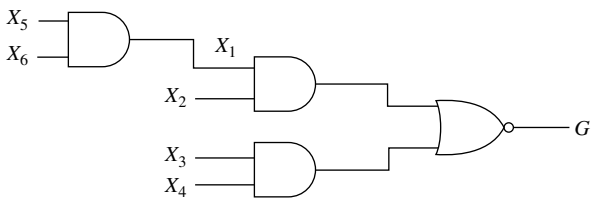


Figure 4.6 AOI with AND gate input.

4.3.4 Justification and Implication

We created a set of inputs for a primitive circuit and saw how to propagate the resulting test through other logic in order to make the test visible at an output. Signal assignments made to the outputs of primitives during the propagation phase must also be justified. Consider the circuit of Figure 4.6. It is the AND-OR-Invert with input 1 now driven by an AND gate. We want to again test input X_1 for the SA1 fault. Therefore input X_1 of the AOI must be 0. Because we are familiar with the behavior of the AND gate, we can easily deduce that either input X_5 or X_6 must be 0 to get the required 0 at X_1. Alternatively, we can go to the cover for the AND gate and select an entry from p_0. The selected entry will tell us what values must be applied to the inputs in order to get the required 0 on the output.

The selected entry may not always be acceptable. In Figure 4.7 we again consider the AOI as a primitive. It is configured as a 2-to-1 multiplexer by virtue of the inverter. If the goal is to create a test for a SA1 on the net labeled X_2, then the first step is to apply (1, 0, 0, X) to nets X_1, X_2, X_3, and X_4. These assignments must be justified. Assuming the 1 on net X_1 can be justified, then the 0 assigned to net X_2 must be justified. When we examine the cover for the inverter, we find that we need a 1 on the input. This requires a 1 on the output of the AND gate. We then seek to justify the 0 on net X_3, but it requires a 0 from the AND gate. A conflict exists. It is obviously not possible to get a 0 and 1 simultaneously from the AND gate.

To resolve this conflict, an alternate decision must be made. Fortunately another PDCF, (1, 0, X, 0), exists for the fault. With this alternate PDCF net, X_3 no longer requires an assignment. The original PDCF (1, 0, 0, X) implied a 0 at the output of the AND gate and hence to the input of the inverter. That in turn implied a 1 on the output of the inverter and produced a conflict. Had the implications of the test (1, 0, 0, X) been extended, the computations required to justify the assignment on net 1 could have been avoided.

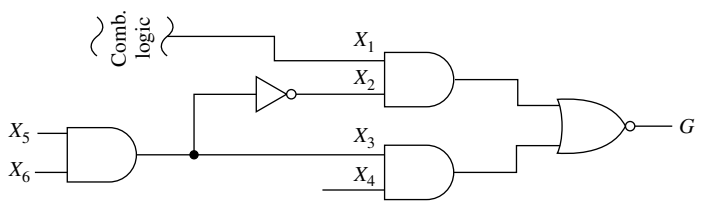


Figure 4.7 AOI as a multiplexer.

4.3.5 The D-Intersection

Covers, PDCFs, and propagation D-cubes have now been developed. These must be used to create tests for circuits composed of numerous interconnected primitives. This will be accomplished by means of the D-intersection that we define with the help of another of our ubiquitous intersection tables.

D-INTERSECTION TABLE

	0	1	X	D	$\overline{D}$
0	0	—	0	—	—
1	—	1	1	—	—
X	0	1	X	D	$\overline{D}$
D	—	—	D	μ	λ
$\overline{D}$	—	—	$\overline{D}$	λ	μ

The D-intersection table defines the results of a pairwise intersection of corresponding elements of two vectors whose elements are members of the set {0, 1, D, $\overline{D}$, X}. The elements represent the values on the inputs of a circuit as well as the values on the outputs of individual primitives in the circuit.The dash (—) indicates a conflict, in which case the intersection does not exist. We postpone discussion of λ and μ until later.

The D-intersections will be used to extend a sensitized path from the point of a fault to the inputs and outputs of the circuit. The first step is to select a fault and assign a PDCF. The propagation D-cubes and the cover are then used in conjunction with the D-intersection table to form subsets of connected nets where we say that two nets are *connected* if the values assigned to them are the direct result of (a) the assignment of a PDCF or (b) a succession of one or more nontrivial D-intersections.

A nontrivial intersection requires that the vectors being intersected have at least one common coordinate position in which neither of them has an X value.

The set of all connected nets forms a subcircuit called the *test cube*, also sometimes called a *D-chain*. Associated with a test cube are an *activity vector* and a *D-frontier*. The activity vector consists of those nets of the test cube that (a) are outputs of the test cube and (b) have a value D or $\overline{D}$ assigned.

The D-frontier is the set of gates with outputs not yet assigned that have one or more input nets contained in the activity vector. The objective is to start with the PDCF and form an expanding test cube via D-intersections between an existing test cube and the propagation D-cubes and members of the primitive covers until the test cube reaches the circuit inputs and outputs.

Example The D-algorithm will be used to create a test for the circuit in Figure 4.3. Operations will be listed in tabular form, numbers will be assigned to relevant steps, and we will refer to the step numbers as we explain the operations. The calculations are shown in Figure 4.8.

	1	2	3	4	5	6	7	8	9	10	11	12
1.	X	0	0	X	X	D	X	X	X	X	X	X
2.	0					D			$\overline{D}$			
	0	0	0	X	X	μ	X	X	$\overline{D}$	X	X	X
3.	0	0	0	X	X	D	X	X	$\overline{D}$	X	X	X
4.				0		D				$\overline{D}$		
	0	0	0	0	X	μ	X	X	$\overline{D}$	$\overline{D}$	X	X
5.	0	0	0	0	X	D	X	X	$\overline{D}$	$\overline{D}$	X	X
6.								0	D	D	0	$\overline{D}$
	0	0	0	0	X	D	X	0	λ	λ	0	$\overline{D}$
5.	0	0	0	0	X	D	X	X	$\overline{D}$	$\overline{D}$	X	X
6.								0	D	D	0	$\overline{D}$
	0	0	0	0	X	D	X	0	μ	μ	0	D
7.	0	0	0	0	X	D	X	0	$\overline{D}$	$\overline{D}$	0	D
8.			X				1				0	
	0	0	0	0	X	D	1	0	$\overline{D}$	$\overline{D}$	0	D
9.		X			1			0				
10.	0	0	0	0	1	D	1	0	$\overline{D}$	$\overline{D}$	0	D

Figure 4.8 D-chain for Schneider's counterexample.

In the first step a PDCF was assigned for a SA0 on the output of gate 6. It was then propagated through gate 9. The intersection produced the result μ on the output of gate 6. We now give the rules for processing the μ and λ symbols:

1. If one or more μs occur, convert them to the corresponding D or $\overline{D}$ signals that appear in the test cube and propagation D-cube.
2. If one or more λs occur, complement all D and $\overline{D}$ signals in the propagation D-cube, perform the intersection again, and convert the resulting μs according to rule 1.
3. if μs and λs both occur, the intersection is null.

In accordance with rule 1, the μ on the output of gate 6 is converted to a D. Because gate 6 fans out to two gates, the activity vector consists of gates 6 and 9 and the D-frontier consists of gates 10 and 12. We refrain from implying signals in this example, choosing instead to propagate through gate 10 in step 4. We again produce a μ which is converted to a D.

In step 6, propagation occurs through gate 12, producing a λ on gates 9 and 10. The D and $\overline{D}$ signals in the propagation D-cube are complemented, and for convenience the step is relabeled as step 6′. This results in μ appearing on gates 9 and 10. These are then both converted to $\overline{D}$ in step 7. In this step a multiple path was propagated through

gate 12. The values at the inputs to gate 12 are (0, 0, 0, 0) for the fault-free circuit and (0, 1, 1, 0) for the faulted circuit. If propagation D-cubes with multiple D and $\overline{D}$ signals are not stored in the propagation D-cube table, it would be necessary to create the required propagation D-cube, using the cover consisting of vertices.

Finally, having propagated a signal to the output, assignments to internal gates must now be justified. In step 8 the assignment of a 0 to gate 11 is justified by assigning a 1 to gate 7 and an X to input 3. In step 9 the same is done for gate 8. It is also necessary to justify the values assigned to gates 7 and 5, but at this stage it merely requires confirming that the values on their inputs satisfy the requirements on the outputs, since there are no more assignments that can be made. The final test cube is shown in line number 10. ■■

Fortunately, it was not necessary to invoke rule 3, μ and λ did not occur simultaneously. If they had, then it indicates that the test cube and the propagation D-cube have D and $\overline{D}$ signals in more than one common position. Furthermore, some of the signals were in agreement and some were in conflict. Therefore, complementing all D and $\overline{D}$ signals in the propagation D-cube will not resolve the conflict.

The D-algorithm is sometimes referred to as a two-dimensional algorithm, in contrast to path sensitization, which has been characterized as one-dimensional. Strictly speaking, the path sensitization method is not even an algorithm, but, rather, a *procedure*. The distinction lies in the fact that an algorithm can always find a solution *if a solution exists*. In other respects they are similar, since both an algorithm and a procedure can be programmed, such that a next step or a criterion for termination always exists. The reader is cautioned to note that authors are not consistent on the usage of these terms, some calling an algorithm that which is more accurately called a procedure. While we may not always strictly adhere to this distinction, the reader should be aware that when an author sets out to demonstrate that his method is an algorithm, he must show that it will find a solution whenever a solution exists.

The proof that the D-algorithm is an algorithm consists of showing that if a test cube $c(T,F)$ exists for failure F, the test cube $c(T,F)$ must be contained in a PDCF. Also, a test cube must contain a connected chain of coordinates having values D or $\overline{D}$ linking the output of the faulted gate to a primary output. Given a particular gate through which the test passes on its way to an output, the test cube $c(T,F)$ must be contained in some propagation cube of the gate in question since the propagation D-cubes are constructed so as to define all possible combinations by which a test can be propagated through the gate. Finally, the fact that all propagation D-cubes are candidates for intersection, including those with multiple propagation paths, assures that all possible chains can be constructed, implying that, given a particular test, the D-algorithm will find that test (if it does not find some other test first).

4.4 TESTDETECT

The D-algorithm is used to construct sensitized paths extending from fault origins to primary outputs. The *D*-notation keeps track of values along the way, and the tables

that define operations on pairs of logic signals and/or *D*-symbols make it possible to evaluate progress, as well as to identify nodes where signals occur that block or impede the progress of the *D*-signals. Using this same *D*-notation, Paul Roth developed a procedure, called *Testdetect*, that relies on *D*-signals to determine which faults are detected by a given input vector.[4]

To understand the operation of Testdetect, consider the circuit in Figure 4.1. The input pattern $I_1, I_2, I_3, I_4, I_5 = (0, 0, 1, 0, 0)$ is applied to the circuit. This input pattern results in a 0 at the output *Z*. Obviously, if the output is SA1, the fault will be detected. The outputs of gates *K*, *L*, *N*, and *O* are all 1s for the fault-free circuit. If the output of any of these gates is SA0, that fault will cause the output to assume the value 1; hence those SA0 faults will also be detected. It is possible to continue tracing back toward the inputs, from any fault that is detected, to identify other faults that will be detected. For example, if an SA0 on the output of gate *L* is detectable, then any fault on the input of *L* that causes its output to assume the value 0 is also detectable.

Testdetect formalizes this approach. It selects a fault and determines whether a D-chain can be extended from this fault to an observable output. However, in this inverse D-algorithm, all signal values are fixed. The objective is not to create a test but rather, having created a test, to determine what other faults are detected by the input vector. Therefore, the object is to determine, for a given fault, if its effects propagate through a series of gates, eventually reaching an output.

A *D-list* keeps track of gates in the D-frontier while progressing toward primary outputs. A gate is selected from the D-list, and it is determined whether the fault will propagate through the gate. If not, then the D-chain has died on that path; and if the D-list is empty, the fault will not be detected by that test vector. If the fault does propagate through the gate, then the gate or gates in the fanout from that gate are placed in the D-list. This continues until either

1. A primary output is encountered, or
2. The D-list becomes empty.

A third criteria for stopping exists:

Lemma 4.1 If at any stage in the computation for failure *F*, the *D*-frontier reduces to a single net *L* and there is no reconvergent fanout beyond the *D*-frontier, then *F* is testable iff if *L* is testable.[5]

Rules for determining whether or not a fault propagates through an element are the same as those used in the D-algorithm. For an AND gate with a D or $\overline{\text{D}}$ on an input (or inputs), if the other inputs are all 1s, then the D or $\overline{\text{D}}$ will propagate to the output of the gate. In general, if the good circuit signal causes a 1 (0) on the output of the gate and the fault causes a 0 (1), then the fault signal propagates to the output of the gate.

Example For the circuit of Figure 4.1, with the inputs $I_1,I_2,I_3,I_4,I_5 = (0,0,1,0,0)$, the output of gate *L* has a 1. An SA0 on the output of *L* produces a D, which shows up at the output of the circuit as a $\overline{\text{D}}$. Hence the SA0 is detected. If the upper input to gate *L* is SA0, then ($\overline{\text{D}}$,0) produces a D on the output of *L*. By the lemma, the fault is detected. However, an SA0 on the output of gate *D* must be analyzed all the way to the output because there are two gates, *J* and *L*, in its D-list.

A D is assigned to the output of gate *D*, indicating a SA0 on its output, and *J* and *L* are placed in the D-list. We assume that the circuit has been rank-ordered, and we require that when there are two or more entries in the D-list, the lower numbered gate to be selected first. (Why?) Therefore, gate *J* is selected for processing. The inputs to gate *J* are (0,0,1,D). Since the 1 on the third input is inverted at the input, the output of *J* is a $\overline{D}$. This causes *K* to be placed in the D-list. Since it precedes *L* (alphabetically), it is processed next. The $\overline{D}$ from gate *J*, together with the 0s on its other inputs, causes a D to appear on its output. Gate *L* is processed next, and a D appears on its output. The subcircuit consisting of *M*, *N*, *O*, and *P* represents an exclusive-OR, so the D signals appearing at the inputs to this subcircuit cancel at the output. Hence the fault on the output of gate 9 is not detected by this test pattern. ■■

The failure to detect a fault on the output of gate *D*, despite the fact that it drives a gate on which faults are detected, is caused by reconvergence of two sensitized paths that cancel each other out. If there were no problems with reconverging logic, Testdetect could run quite rapidly and work straight from the outputs back to the inputs. However, reconvergent fanout necessitates that all fanout branches be examined. In the example, we looked at a situation where a pair of D-chains diverged at the D-frontier. It is possible to have a D-frontier with a single element that is detectable and still not have a detectable fault. Such a condition is illustrated in Figure 4.9.

With the input combination 1, 2, 3 = (1, 0, 0), a fault on the output of gate 5 is detectable. But, consider what happens if the input combination 1, 2, 3 = (1, $\overline{D}$, 0) is applied to test for an SA1 at input 2. This causes a D to appear at the output of gate 5 and causes a $\overline{D}$ to appear at the output of gate 4. With D and $\overline{D}$ on its inputs, the output of gate 6 is a 0. We are left with only gate 5 in the D-list, and that was previously determined to be detectable by the applied pattern, yet the SA1 at primary input 2 is not detectable because the 0 on the output of gate 6 prevents the D at gate 5 from reaching the output.

4.5 THE SUBSCRIPTED D-ALGORITHM

Given an AND gate or an OR gate, for each input fault to be tested the D-algorithm must recompute a propagation path from that gate to a primary output. This effort becomes increasingly redundant for circuits in which many gates have a large number of inputs. Elimination of these redundant computations is one of the objectives of the subscripted D-algorithm, or A-algorithm (AALG).[6]

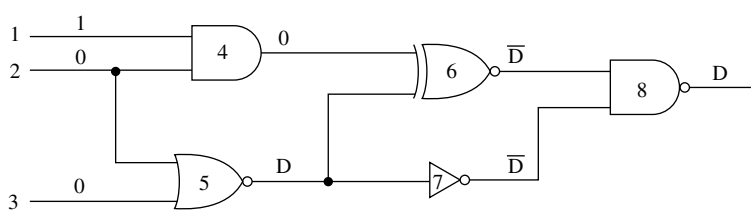


Figure 4.9 Recombining sensitized paths.

The AALG goes farther, however. Whereas the D-algorithm selects a single fault and justifies fixed binary values on the inputs of the corresponding gate, AALG simultaneously justifies symbolic assignments on all inputs in a process called *back-propagation*. The first step in this process is to select a gate and assign the symbol D_0 to its output. This symbol is propagated to a primary output using the same forward-propagation techniques employed in the D-algorithm. If the gate has m inputs, then a symbol D_i, $1 \le i \le m$, is assigned to each of its inputs.The D_i are called *flexible signals*; they may represent 0 or 1, depending on what values are required for a particular test.

After the D_0 signal has been successfully propagated to an output, all of the D_i are back-propagated to primary inputs. If the back-propagation is completely successful, then tests for the output fault and all of the gate input faults can be computed simply by inspecting values at the primary inputs. This is illustrated in the circuit of Figure 4.10, where the input vector I has value $I = (X, 0, D_1, \overline{D}_2, 0, 0)$.

This vector is interpreted by referring back to the gate where the D_i originated. A test for the output of gate 16 SA0 requires both of its inputs to be 1, that is, D_1, $D_2 = (1, 1)$, which requires inputs 3, 4 = (1, 0). Tests for SA1 on inputs 1 and 2 of gate 16 require D_1, $D_2 = (0, 1)$ and (1, 0), respectively. Therefore, the tests for these three faults are

$$(X, 0, 1, 0, 0, 0)$$

$$(X, 0, 0, 0, 0, 0)$$

$$(X, 0, 1, 1, 0, 0)$$

The input assignments are not unique. For example, the input vector I could have been assigned the values $(D_1, 1, X, \overline{D}_2, 1, 1)$. Several other possibilities exist, depending on choices made at gates where decisions were required during back-propagation.

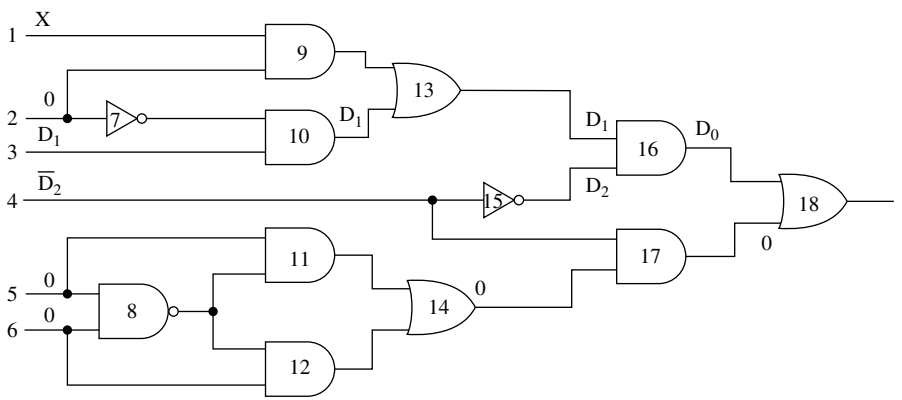


Figure 4.10 Illustrating the subscripted D-algorithm.

We now discuss the rules for back-propagation. Basically, each D_i is back-propagated toward the inputs along as many paths as possible. This is done through replication. When symbolically propagating back through an element, the symbol D_i at the output is replicated at the inputs, according to the following rules:

1. If a gate inverts a signal, then the inputs are assigned $\overline{D}_i$.
2. D_i (or $\overline{D}_i$) is replicated at all inputs if no input has been previously assigned.
3. D_i can be replicated at some inputs if all others are assigned noncontrolling values.

Example Given a three-input NAND gate, with one of its inputs assigned a logic 1, and D_j assigned to its output during back-propagation, the remaining two inputs are assigned $\overline{D}_j$. ■■

This proliferation of D_i signals enhances the likelihood of establishing a sensitized path from one or more primary inputs to input i of the gate presently being tested, in contrast to propagation of a single replica, which may require considerable backtracking* in response to conflicts. However, it is still possible to encounter conflicts. In fact, with flexible signals increasing exponentially in number as progress continues toward the inputs, conflicts are virtually inevitable in any realistic circuit. Efficient handling of conflicts is imperative if performance is to be realized.

A conflict can occur during back-propagation as a result of a signal D_i and a conflicting value of that same signal attempting to control a gate, or as a result of two different signals D_i and D_j attempting to control a gate, or a conflict may occur at a gate with fanout if two or more signal paths reconverge at the gate and one of the paths has a flexible signal while another has a fixed binary value.

The situation in which conflicting values of the same flexible signal try to control a gate is illustrated in the upper path of Figure 4.10. The assignment of D_1 on the output of gate 13 during back-propagation initially results in the replication of D_1 on each of its inputs, hence on the outputs of gates 9 and 10. Back-propagation then produces replicas of D_1 on both inputs of gates 9 and 10. However, we are now faced with the prospect of flexible signal D_1 on both the input and output of inverter I_7. This conflict can be resolved by assigning a 0 or 1 to the output of gate 7. Choosing a 1 forces 0s on the input of gate 7 and the lower input of gate 9, which forces a 0 on the output of gate 9 and also causes the upper input to gate 9 to be reassigned to X.

The conflict between flexible signals D_j and D_k can be illustrated by assigning D_0 to gate 14. Forward propagation and justification along the upper path are the same as in the D-algorithm. We therefore restrict our attention to the consequences of a D_0 on gate 14. This requires D_1 and D_2 on the inputs to gate 14. Back-propagation then attempts to assign both D_1 and D_2 to the output of gate 8. Again, the conflict is

*In the discussion that follows, the terms backtracing and backtracking will be used. It is easy to confuse them. Backtracing is the process of working backward in the circuit model, while backtracking is the process of correcting for a conflict between node values.[7]

resolved by assigning a fixed binary value to the output of gate 8. If a 1 is assigned, then one of the inputs must be set to 0. However, the other flexible signal can still be instantiated.

Generally, when an input must be set to a controlling value—for example, a 0 on an input to an AND or NAND gate—it is usually preferable to choose the input that is easiest to control. However, in the present case an additional criterion may exist. If a fault on one of the two inputs to gate 14 has already been detected, then the flexible signal D_1 or D_2 corresponding to the undetected input fault can be favored when a choice must be made. When D_1 and D_2 converge at the output of gate 8, if it is found that the upper input to gate 14 has already been tested, then D_1 can be purged by assigning a 0 to the upper input of gate 8.

When a conflict occurs, its resolution usually requires that segments of D_i chains be deleted. AALG accomplishes this with functions called DROPIT and DRBACK.[8] DROPIT purges a chain segment when the end closest to the primary inputs is known. It works forward toward the gate under test. It must examine fanouts as it progresses, so if two converging paths both have flexible signals, then both chain segments must be deleted. When a flexible signal is deleted, it may be replaced by a fixed binary signal. This signal, when assigned to the input of a gate, may be a controlling value for that gate and thus implies a logic value on the output. In that case, the output must be further traced to the input of the gate(s) in its fanout to determine whether this output value is a controlling value at the input of the gate in its fanout.

When D_0 was assigned to the output of gate 14, a conflict occurred at gate 8, so a 1 was assigned to its output, which required a 0 on one of its inputs. Primary input 6 was chosen. This required that the D_2 chain from P.I. 6 to the input of gate 14 be purged. A 0 on P.I. 6 implies a 0 on the output of gate 12, so the flexible signal D_2 initially assigned at the output of gate 12 must be purged and the path traced another level. At gate 14 the enabling signal 0 is assigned to the lower input and the flexible signal D_1 is assigned to the upper input. Therefore DROPIT can stop at that point.

If D_j controls the output and one or more D_i control the inputs, it may be desirable to propagate D_j toward the inputs and purge the D_i signals. In that case the end of the chain farthest from the PIs is known and DRBACK purges the chain. Working back toward the PIs, it may have to purge a considerable number of flexible signals since the signals were originally replicated when working toward the inputs.

The functions DROPIT and DRBACK are not always invoked independently of one another. When DROPIT is purging flexible signals and replacing them with fixed binary signals, it may be necessary to invoke DRBACK to purge other chain segments. This is seen in the upper branch of the circuit in Figure 4.10. Primary input 2 was assigned a 0 because of a conflict. Therefore DROPIT, working forward from primary input 2, purges D_1 and replaces it with a 0. The 0 on the lower input of gate 9 blocks the gate and therefore DRBACK must pick up the chain segment on the upper input and delete it back to input 1 and replace it with X. Then DROPIT regains control and proceeds forward. The 0 on the input of gate 7 implies a 0 on the output and hence a 0 on the input to gate 13. Since a 0 on an OR gate is not a controlling value, the forward purge can stop, leaving gate 13 with $(0, D_1)$ on its inputs.

To help identify and purge unwanted chain segments, flexible signals are never implied forward to primary outputs during back-propagation. As an example, in Figure 4.10, when back-propagating from gate 9 toward primary inputs, any assignment to primary input 2 will necessarily imply the inverse signal on the output of gate 7. However, if the flexible signal is assigned, then at some later point DROPIT may go unnecessarily along signal paths, deleting flexible signals and replacing them with controlling logic values where it may be unnecessary.

In measurements of performance, it has been found that AALG creates an input pattern with flexible signals in about the same time that the D-algorithm generates a single pattern. Overall time comparison for typical circuits shows that it frequently processes a circuit in about 30% of the time required by the D-algorithm. AALG is especially efficient, for reasons explained earlier, when working on circuits that have gates with large numbers of inputs, as is sometimes the case with programmable logic arrays (PLAs). The efficiency of AALG can be enhanced by first selecting primary outputs and then selecting gates with large numbers of inputs. Gates for which the output has not yet been tested are chosen next since they usually indicate regions where fault processing has not yet occurred. Finally, scattered faults are processed. On those faults AALG occasionally defaults to the conventional D-algorithm.

4.6 PODEM

The D-algorithm selects a fault from within a circuit and works outward from that fault back to primary inputs and forward to primary outputs, propagating, justifying and implicating logic assignments along the way. In circuits that rely heavily on reconvergent fanout, such as parity checkers and error detection and correction (EDAC) circuits, the D-algorithm may encounter a significant number of conflicting assignments. When that happens it must find a node where an arbitrary choice was made and choose an alternate assignment. This can be very CPU and/or memory intensive, depending on how many conflicts occur and how they are handled.

PODEM (path-oriented decision making)[9] reduces the number of remade decisions by selecting a fault and assigning logic values directly at the circuit inputs to create a test. Much of its efficiency results from its ability to exploit the fact that signal polarity along sensitized paths is irrelevant. For example, when the D-algorithm propagates a D or $\overline{D}$ through an XOR, it assigns a 1 or 0 to the other input, the choice being arbitrary and often depending on how the software was coded. It may then go to great lengths to justify that choice, despite the fact that either choice is equally effective, and the chosen value may eventually produce a conflict, necessitating a remade decision. PODEM, as we shall see, implicitly propagates through the XOR, eliminating the need to make a choice at the other input, thus obviating the need to make or alter a decision.

PODEM begins by initializing the circuit to Xs. A fault is chosen, and PODEM backs up through the logic until it arrives at a primary input, where it assigns a binary value, 0 or 1. Implications of this assignment are propagated forward. If either of the following propositions is true, the assignment is rejected.

1. The net for the selected stuck fault has the same logic value as the stuck fault.
2. There is no signal path from an internal net to a primary output such that the internal net has value D or $\overline{D}$ and all other nets on the signal path are at X.

Proposition 1 excludes input combinations that cause the fault-free circuit to assume the same value as the stuck-at value at the site of the fault. Proposition 2 rejects input combinations that block all possible paths from the fault to the outputs. If the test is not complete and if there is no path to an output that is free to be assigned, then there is no way to propagate a test to an output.

When PODEM makes assignments to primary inputs, it employs a *branch-and-bound* method.[10] This process is represented by the tree illustrated in Figure 4.11. An assignment is made to a primary input and is implied forward. If the assignment does not violate proposition 1 or 2, it is retained and a branch is added to the tree. If a violation occurs, the assignment is rejected and the node is flagged to indicate that one value had been unsuccessfully tried. The tree is thus bounded. If the node had been previously flagged, then it is completely rejected and it becomes necessary to back up in the tree until an unflagged node is encountered, at which point the alternate value is implied. The process continues until a successful test is created or the process returns to the start node and both choices have been tried. If that occurs, it is concluded that a test does not exist. The criterion for a successful test is the same as that employed by the D-algorithm, namely, that a D or $\overline{D}$ has propagated from the point of a fault to a primary output.

If PODEM rejects the initial assignment to the ith input selected, and if there are n primary inputs, then 2^{n-i} combinations have been eliminated from further consideration. If the initial assignment to the first primary input is rejected, then the number of

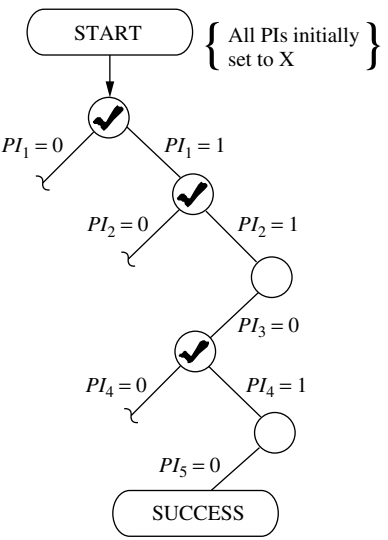


Figure 4.11 Branch-and-bound without backtrace.

combinations to be considered has been cut in half. We say, therefore, that PODEM examines all input combinations implicitly. It does not have to explicitly evaluate all assignments in order to determine if a test exists. Since it will consider all possible input combinations if necessary to find a test, it can be concluded that if PODEM does not find a test, a test does not exist; hence it follows that PODEM is an algorithm.

PODEM can be implemented by means of a last-in, first-out (LIFO) stack. As primary inputs are selected, they are placed on the stack. A node is flagged if the initial assignment was rejected and the alternate choice is being tried. If a node assignment violates one of the two propositions and the node is flagged, then the node is popped off the stack, thus bounding the graph. Nodes continue to be popped off until an unflagged node is encountered. The process terminates when a test is found or the stack becomes empty.

Example The branch-and-bound method is illustrated in Figure 4.11, corresponding to an SA0 on input 3 of gate K of the circuit in Figure 4.1. In this example, the initial trial assignments are arbitrarily chosen to be 0. When a 0 is assigned to I_1 a problem occurs immediately because the output of gate H becomes 0, and that violates rule 1 above. Therefore the assignment is rejected and the alternate value is assigned. The initial assignment to I_2 is rejected for the same reason. The assignment $I_3 = 0$ is retained, at least for the moment, because it does not violate either of the two rules.

The next assignment, $I_4 = 0$, has to be rejected because it causes the output of gate C to become 0, which causes the output of gate H to become 0, again violating rule 1. The assignment $I_4 = 1$ does not violate either of the rules, so it is retained. Finally, the assignment $I_5 = 0$ completes the test. ■■

PODEM uses the branch-and-bound technique, but its performance is improved substantially by the use of a backtrace feature. The backtrace starts at the gate under test or at some other gate along the propagation path and determines an *initial objective*. The initial objective is a net value and logic value (n, e), $e \in \{0,1\}$, that satisfy the value at the net, either helping to propagate a fault from the input to the output of the faulted gate or helping to extend a sensitized path from the fault origin to an output.

With an initial objective as its starting point, backtrace works back to the primary inputs. During processing, backtrace may encounter a gate such as an AND where all inputs must be set to noncontrolling values. If that happens, it processes the inputs in order, from the most difficult to the least difficult to control. If the backtrace encounters a gate where it is necessary to set an input to the controlling state—for example, a 1 on an input to an OR gate—it chooses the input that is easiest to control to the desired value.

Example Consider again the circuit in Figure 4.1. For the SA0 on input 3 of gate K, the output of gate F must be 0, so one of its inputs must be 1. If the top input is chosen, the 1 comes from inverter A, which requires that I_1 be 0. Implying this assignment causes the output of gate H to become 0. Since gate H drives the third input to K, which is being tested for a SA0 fault, that input must be a 1. This conflict necessitates that primary input I_1 be set to 1, which implies a 0 on the output of gate A.

Since I_1 is set to 1, the top input to K remains unassigned, so another backtrace must be performed from that input, but values implied by the logic 1 on I_1 must not be altered. Therefore, the 0 on the output of gate F is justified this time by a 1 on input I_2. The second input to K also requires a 0, which is required from gate G. But that value is satisfied at this point by the 0 at the output of gate A. The third input to K, the input being tested for a SA0 fault, must be set to 1. A backtrace from that input may encounter gate B or C, both of which must provide a 1. Assume that gate B is processed first. Gate B equals 1 only if one of its inputs is 0, so set I_3 to 0. At this point, gate C is still at X. To get a 1 from gate C requires another backtrace, which causes input I_4 to be set to 1.

The sensitized path must now be propagated forward to the output. If the circuit is rank-ordered and if the rule is to drive the fault to the highest numbered gate, using the crude metric that the highest numbered gate is closest to an output, then gate N is chosen for propagation. With the sensitized signal on the upper input to gate N, the lower input to N must be a 1. Since K has the test signal $\overline{D}$, it is necessary to get a 0 from gate L. The upper input to L has a 0, and $I_4 = 1$, so the backtrace chooses I_5 to be 0. ■■

The backtrace operation determines which primary inputs are relevant when testing a given fault. Furthermore, the backtrace often, but not always, chooses the correct value as the initial trial value for the branch-and-bound operation. A smart backtrace—that is, one that uses clever heuristics—can reduce the number of backtracks needed on the primary inputs. This will be seen in Section 4.7, which discusses the FAN algorithm. The algorithm for PODEM is described below in pseudo-C-code; that is, it follows the C programming language syntax for loop control. For example, in C the expression

```
for(;;) { ... one or more lines of code ... }
```

represents an infinite loop. The only way out is to perform a break somewhere in the code. The open parentheses and close parentheses ({}) are used in lieu of *begin* and *end* to demark a block of two or more lines of code, and they are used to denote a set or collection of objects. For example, {primary inputs} denotes a set of primary inputs. Which primary inputs are being referred to will be evident from the context. Also, two consecutive equal signs (==) indicate a comparison. Note that the backtrace routine searches for an X-path. That is a path from the D-frontier to a primary output which has the value X along its entire length.

```
PODEM()  // call with gate no. and stuck-pin number

{
  for(;;) {
    status = backtrace();       // returns FAIL or P.I.
    if (status == FAIL) {       // back up on input
                                // assignments
      for(;;) {                 // loop through P.I.s
```

```
        if (decision_stack == EMPTY)
          return(FAIL);          //no more P.I.s,
                                 //undetectable fault
        else if (decision_stack.flag == 0) {  //try alt.
                                              value
          P.I.[j] = - P.I.[j];  //complement the
                                 //assignment
          decision_stack.flag = 1;
          break;
        }
        else {                   // back up
          P.I.[j] = X;
          decision_stack.flag = 0;
          pop decision_stack;
        }
      }
    }
//either fall-through or come here after
//returning from backtrace(), i.e., status == P.I.
    imply P.I.s;
    if (TEST == success) //D or DBAR reached P.O.
      return (TEST);  //return with test vector
  }
}

backtrace()    //initial objective
{
  if (G.U.T. output != X) { //gate under test
    for(;;) {               //loop through D-frontier
      choose gate B in D-frontier closest to an output;
      if (gate == NULL)    //either D-frontier is empty,
        return(FAIL);     //or no X-path to an output
                          //exists
      else if (X-path exists from B to output){
          //propagate
        set output of B to 1(0) if AND/NOR(NAND/OR);
        break;
      }
      else continue;    //check next entry in D-frontier
    }
  }
  else { //output of G.U.T. is X
```

```
    if (stuck fault is on G.U.T. input pin) {
      if (faulted input == X)
        faulted input = -(stuck-fault direction);
      else   //propagate value
        set G.U.T. output to 1(0) if G.U.T. is AND/NOR
          (NAND/OR);
    }
    else
      G.U.T. output = -(stuck-fault value); // complement
  }
  for(;;) { //work back until a P.I. is reached
    if (objective net driven by P.I.[j])
      return(P.I.[j]);                //reached a P.I.
    else {  //objective net is driven by gate Q
      if ((OR/NAND and C_O == 1) or (AND/NOR and C_O == 0))
        choose new objective net n;   //input to Q
//      n = X, and EASIEST to control
      else
// ((OR/NAND and C_O == 0) or (AND/NOR and C_O == 1))
        choose new objective net n;   //input to Q
//      n = X, and HARDEST to control
    }
    if (Q == NAND/NOR)       //complement the current
                             //objective level
        objective level = -(C_O logic level);
    else                     //Q is AND/OR
         objective level = C_O logic level;
  }
}
```

4.7 FAN

FAN[11] (fanout-oriented test generation algorithm), like PODEM, uses implicit enumeration. However, it employs a number of additional features designed both to reduce the number of backtracks and to minimize the amount of processing during each backtrack. Some of the more significant enhancements include:

- Maximum use of implication, forward and back
- Multiple backtrace

- Unique sensitization
- Stop at head lines
- Seek consistency at fanout points

PODEM assigns binary values to primary inputs and implies them forward. By way of contrast, FAN implies assignments in both directions to the fullest extent possible in order to more quickly detect conflicts. Consider the circuit in Figure 4.1. Suppose the bottom input of gate G is SA1. The PDCF is (1,1, 0, 0) (note that the bubble on input 3 represents a signal inversion). When all implications, forward and back, of that PDCF are carried out, the fault is immediately seen to be undetectable. However, PODEM may perform several computations, even on this small circuit, before it concludes that the fault is undetectable. These faults cause ATPG programs to expend a lot of useless computational effort because many possibilities frequently must be explored before it can be concluded that the fault is undetectable. If a circuit has many undetectable faults, the ATPG may expend half or more of its CPU time attempting to create tests for these faults. Efficient operation of an ATPG dictates that undetectable faults be found as quickly as possible.

The *multiple backtrace* enables FAN to reduce the number of backtraces and more quickly identify conflicts. Consider again the circuit in Figure 4.1. When justifying a 1 on the third input of gate *K*, PODEM used two backtraces: The first backtrace set I_3 to 0, and the second backtrace set I_4 to 1. When FAN is backtracing, it recognizes that a 1 on the output of gate *H* requires that all of its inputs be at 1, so those values are immediately assigned to its inputs. Any assignment that conflicts with those assignments is immediately recognized. In addition, the backtrace from the third input of *K* to the inputs of *H* are avoided.

The PODEM algorithm, as published, chooses the input that is most difficult to control if all inputs must be assigned noncontrolling values. The reason for choosing the most difficult assignment is that if there is a problem, or conflict, that choice is usually most likely to reveal the conflict as quickly as possible. However, PODEM only assigns the input that is most difficult to control. Thus, if a three-input AND gate requires 1s on all inputs, and all inputs are driven by primary inputs, PODEM will backtrace three times. The multiple backtrace assigns 1s to all three inputs immediately.

The *unique sensitization* operation is performed whenever the D-frontier consists of a single gate. Consider the circuit in Figure 4.12. AND gate *G* is being tested for a SA1 fault on its upper input. The fault must propagate through the multiplexer and then through AND gate *H*. In order for the fault effect to get through gate *H*, its upper input must be 1. But, when setting up the PDCF, it is possible that the upper input to *H* was set to its blocking value. A lot of unnecessary computations might be performed before that conflict is revealed. FAN searches forward along the propagation path to an output searching for these situations. Note that the fault propagates through the select line of the mux, which enters reconvergent logic, so nothing can be said about the logic inside that function. When a situation such as that which exists at gate *H* is encountered, the nonblocking value, in this case the logic value 1, is implicated back toward the primary inputs. The values on the primary inputs must establish a 0 on the faulted input to *G*, and at the same time they must establish a 1 on the upper input of *H*.

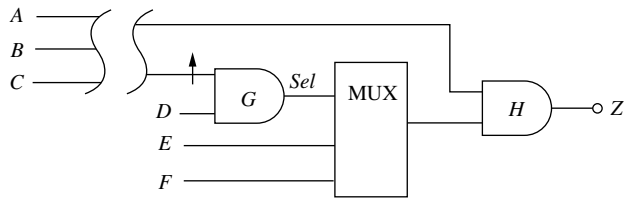


Figure 4.12 Unique sensitization.

Backtracing in FAN is aided by the observation that fanout-free regions (FFRs) usually exist in the circuit being tested. FFRs are single-output subcircuits that do not contain reconvergent logic; hence they can be justified without concern for conflicts. As a result, a backtrace can stop at the outputs of the FFRs. After all other assignments have been made, justification of the FFRs can be performed. This can be seen in the circuit in Figure 4.13, which will be used to help define some terminology.

When a net drives two or more gates, the part of the net common to every branch is called a *fanout point*. In Figure 4.13 the segment J, which is common to J_1 and J_2, is a fanout point. (In this circuit, except for fanout branches, nets will be identified with the gates that drive them.) If a path exists from a fanout point forward to a net P, then P is said to be *bound*. A net that is not bound is *free*. In Figure 4.14 the nets *A, B, C, D, E, F, G, H, I*, and J are free nets, and the nets J_1, J_2, K, and L are bound nets. Note that the net connecting the output of gate J to gates K and L has three identifiable segments: segment J, which is the fanout point; segment J_1, which drives gate K; and segment J_2, which drives gate L. Free nets that drive bound nets, either directly, as in the case of the fanout point J, or through a logic gate, as in the case of K, are called *head lines*; they define a boundary between free lines and bound lines.

The FAN algorithm works with objectives. These are logic assignments that must be satisfied during the search for a test solution. A backtrace in FAN begins with *initial objectives*. At the start of the algorithm initial objectives are determined by the

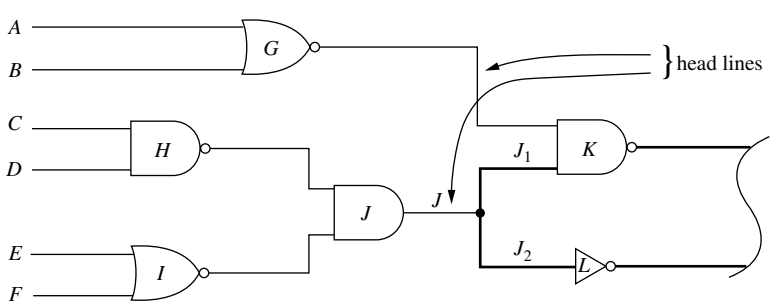


Figure 4.13 Identifying head lines.

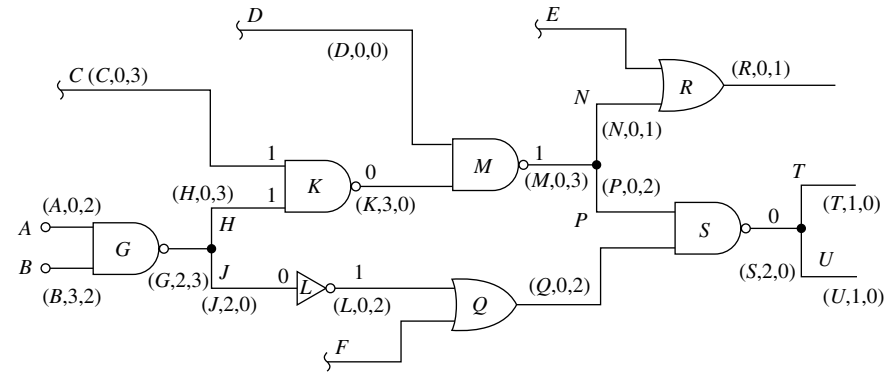


Figure 4.14 Identifying/resolving a conflict.

PDCF. The initial objectives become *current objectives* upon entering the routine, denoted Mback, that performs the multiple backtrace. During the backtrace, logic assignments are made in response to current objectives. These assignments become new current objectives, or they may become head objectives or fanout point objectives, which must eventually be satisfied. Objectives that occur at head lines are called *head objectives*. Objectives at fanout points are called *fanout point objectives* (FPOs).

While assigning logic values to justify current objectives during backtrace, FAN stops at fanout points and head lines until all current objectives have been satisfied. Then the backtrace selects an FPO closest to the primary output, if one exists. Head objectives are always satisfied last, after all other objectives have been satisfied, since there is no reconvergent fanout and they can be satisfied without fear of conflict. If the FPO has conflicting requirements, the conflict must be resolved. A conflict occurs if, during the multiple backtrace, two or more paths converge on the fanout point with different requirements. If the FPO does not require conflicting assignments, the MBack routine continues from this FPO.

In order to maintain a record of logic values that must be assigned during backtrace, as well as to recognize conflicts, FAN employs an objective expressed as a triplet $(s, n_0(s), n_1(s))$. In this triplet, s denotes the objective net, $n_0(s)$ is the number of times a 0 is required at s during the backtrace, and $n_1(s)$ is the number of times a 1 is required at s. A conflict exists if both $n_0(A_i)$ and $n_1(A_i)$ are nonzero. If a conflict exists, the rule is: If $n_0(A) < n_1(A)$, assign a 1 to the fanout point, otherwise assign a 0.

Logic values assigned during backtrace depend on (a) the function of the logic gate through which the backtrace passes and (b) the value required at the output of that gate. For an AND/NAND gate, a 1/0 on the output requires 1s on all inputs. For an OR/NOR gate, a 0/1 on the output requires 0s on all inputs. In addition, if the output is complemented, then the values n_0 and n_1 are reversed in the triplet. For example, given a NOR gate with triplet (Z, u, v) at its output, the triplet assigned to each of its inputs X_i is (X_i, v, u) if a 1 is needed at the output.

If a controlling value is required on the input of a gate (0 on an AND or NAND gate, 1 on an OR or NOR gate), then the backtrace is made through the input that is easiest to control. Assume a logic gate with inputs X_1 ..., X_n, and output Y, and, without loss of generality, assume that input X_1 is the easiest input to control. Then Table 4.1 contains the criteria used to compute the values n_0 and n_1 at each input net.

Consider the AND gate: If a 0 is required at its output, then a 0 must be applied to one of its inputs. Assign a 0 to the input that is easiest to control, unless that input has already been tried and rejected. The values $n_0(X_1)$ and $n_1(X_1)$ at that input are equal to the value at the output. For noncontrolling inputs we have $n_0(X_i) = 0$ and $n_1(X_i) = n_1(Y)$. Similar considerations hold for the NAND gate except that from Table 4.1 it can be seen that the subscripts are reversed. The analysis for the OR and NOR gates are similar, but complementary.

At FPOs the values n_0 and n_1 are summed. This is in recognition of the fact that, during backtrace, two or more paths driven by that FPO may have requirements to justify signals further along toward the output. Furthermore, if two or more nets require the same value from an FPO, by summing their requirements, it is possible to determine how many signal paths depend on each value, 0 or 1, generated by that FPO.

These computations can be illustrated using the circuit in Figure 4.13. Assume the values (J_1,1,1) and (J_2,1,2) occur at segments J_1 and J_2 during backtrace in order to justify assignments made closer to the output. The value 0 has weight 2, and the value 1 has weight 3. When this happens, the logic value 1 is chosen to be assigned at the FPO. But, since that represents a conflict, the multiple backtrace is halted at this point and conflict resolution is performed. That involves backtracking on assignments made to the FPO and trying alternate assignments. If a self-consistent set of assignments to the FPOs cannot be found, the fault is undetectable.

TABLE 4.1 Assignment Criteria

	Function	0-count	1-count	Controllability
1	AND	$n_0(X_1) = n_0(Y)$	$n_1(X_1) = n_1(Y)$	Easiest 0
2	AND	$n_0(X_i) = 0$	$n_1(X_i) = n_1(Y)$	Others
3	NAND	$n_0(X_1) = n_1(Y)$	$n_1(X_1) = n_0(Y)$	Easiest 0
4	NAND	$n_0(X_i) = 0$	$n_1(X_i) = n_0(Y)$	Others
5	OR	$n_0(X_1) = n_0(Y)$	$n_1(X_1) = n_1(Y)$	Easiest 1
6	OR	$n_0(X_i) = n_0(Y)$	$n_1(X_i) = 0$	Others
7	NOR	$n_0(X_1) = n_1(Y)$	$n_1(X_1) = n_0(Y)$	Easiest 1
8	NOR	$n_0(X_i) = n_1(Y)$	$n_1(X_i) = 0$	Others
9	NOT	$n_0(X) = n_1(Y)$	$n_1(X) = n_0(Y)$	
10	Fanout	$n_0(X) = \sum_{i=1}^{k} n_0(X_i)$	$n_1(X) = \sum_{i=1}^{k} n_1(X_i)$	

Example The circuit in Figure 4.14 will be used to illustrate the operation of FAN. In this circuit, inputs *A* and *B* are primary inputs, while *C*, *D*, *E,* and *F* are inputs from other parts of the circuit and, where choices must be made, we will assume that *C*, *D*, *E*, and *F* are the more difficult choices. Calculations are summarized in Table 4.2. The example starts with objectives at the nets *R*, *T,* and *U*. The values on nets *T* and *U* are summed to give the value (*S*,2,0) at net *S*. Likewise, the triplets at *N* and *P* are summed to yield the triplet (*M*,0,3). This requires a 0 on one of the inputs to *M* and, for sake of illustration, we assume that net *K* is the easiest to control. Because *M* is a NAND, the values n_0 and n_1 of the triplet at *K* are reversed. Eventually, the fanout point *G* is reached, but with conflicting requirements. Since segment *H* has a higher weight, a 1 is assigned to fanout point *G*. Since G is a headline, assignments to *A* and *B* are postponed.

Because *G* has conflicting requirements, the function MBack is exited and FAN implies the value 1 that was assigned to *G*. The assignment conflicts with the requirement at *L*. That requirement comes from net *Q*, whose objective is (*Q*,0,2). But that objective might be satisfied by the unidentified logic driven by net *F,* in which case the conflict at *G* is resolved. If, however, the conflict cannot be resolved, the alternate value, 0, is assigned to *G*. The conflict along that path can be resolved by assigning a 0 to net *D*. All affected triplets must then be recomputed. Then MBack selects an FPO from which it backtraces in order to obtain and satisfy new current objectives. ■■

We leave it to the reader to complete this example. The FAN algorithm is described in pseudo-C-code at the end of this section.

The first step in FAN is to assign a PDCF for the fault. Then, a backtrace flag is set. The flag enables MBack to distinguish between those instances where a backtrace starts from a set of initial objectives (IO), entry A, or from a set of fanout point objectives (FPO), entry B. Entry B to the backtrace routine is entered in order to continue a multiple backtrace that terminated at a fanout point.

TABLE 4.2 Keeping Track of Objectives

Current Objectives	Stem Obj.	Head Obj.
(*R*,0,1), (*T*,1,0), (*U*,1,0)		
(*T*,1,0), (*U*,1,0), (*N*,0,1)		
(*U*,1,0), (*N*,0,1)	(*S*,1,0)	
(*N*,0,1)	(*S*,2,0)	
	(*S*,2,0), (*M*,0,1)	
(*P*,0,2), (*Q*,0,2)	(*M*,0,1)	
(*Q*,0,2)	(*M*,0,3)	
(*L*,0,2)	(*M*,0,3)	
(*J*,2,0)	(*M*,0,3)	
	(*G*,2,0)	
(*K*,3,0)	(*G*,2,0)	
(*H*,0,3)	(*G*,2,0)	
	(*G*,2,3)	
(*F*,0,2)		(*G*,2,3)

A sensitized value, D or $\overline{D}$, results either from a stuck-fault on the output of a gate, or from a stuck-fault on the input of the gate, in which case it is implied to the output of the gate. The sensitized value continues to be propagated forward from there. If the output of the faulted gate only drives a single destination gate, then the sensitized signal can be propagated to the output of that gate, with the result that additional nonblocking assignments on the input of that gate are added to the set of initial objectives. If the D-frontier consists of two or more entries, FAN examines the entries in the D-frontier to ensure that they are all legitimate; that is, they all propagate to output pins and are not blocked. Then FAN orders these paths in terms of ease or difficulty of propagation. However, like the D-algorithm, an implementation in FAN must, if necessary, eventually consider all single and multiple propagation paths at FPOs to truly be considered an algorithm.

The MBack routine has two entries. At entry A the initial objectives become the set of current objectives {CO}. If {CO} is non-empty, then an objective is selected. While MBack traces back through the circuit, if it encounters a head line, that head line is added to the set of head objectives {HO}. If it encounters a logic gate, then it must be determined if the gate requires a controlling or noncontrolling value on its inputs. As previously discussed, the rules in Table 4.1 are used to select an input and a value to be assigned to that input. The net driving the input is added to the set of current objectives. If the net is a fanout branch, then n_0 and n_1 are updated. However, fanout points are not processed until all of the nonfanout gates are justified.

The other entry to MBack is entry B. This entry is used if the set of current objectives is empty, then an FPO is selected from the set {FPO}. If there is no conflict, MBack continues from the FPO. However, if the node has conflicting requirements, then the conflict has to be resolved. This is accomplished by means of a backtrack through the FPO assignments.

Initially the backtrace flag is on if there are unjustified nets at the completion of the implication stage. At this point all sets of objectives are initialized to empty (EMPTY) and the backtrace flag is reset. If there are unjustified lines, they become the set of initial objectives {IO}. If the error signal did not reach a primary output, a gate in the D-frontier is added to {IO}. A multiple backtrace is then performed by the MBack function. If the backtrace flag is not on, then there are no nets waiting for logic assignments. In that case, the set of fanout point objectives {FPO} are examined. If the set is nonempty, then a multiple backtrace is performed from a selected FPO. At the completion of the multiple backtrace, if there are no conflicts at any fanout points, then the set of header objectives {HO} are processed. If there is a conflict at a fanout point—that is, both $n_0(f)$ and $n_1(f)$ are nonzero—then the value assigned is based on which value is larger. Since both values are nonzero, there is obviously a conflict that must be resolved. Looking again at the final_objective function, a value is assigned and a return is made to the implication step, where a conflict leads to block 8.

```
FAN()   //call with gate no. and stuck-pin number
{
  assign PDCF;                  //primitive D-cube of
                                //failure
```

```
  backtrace_flag = A;          //backtrace from
                               //unjustified lines
  for(;;)  //loop forever
  {
    implicate assignments;   //forward and back
    if (backtrace unnecessary)
      backtrace_flag = B;     //process FPO
    if (fault signal reached a P.O.) {
      if (# unjustified bound lines == 0) {
        justify free lines;  //done
        return (TEST);
      }
      else {
        final_objective();
        assign value to final objective line;
      }
    }
    else {
      if (# gates in D-frontier > 1) { //choose gate
                                       //closest to P.O.
        final_objective();
        assign value to final objective line;
      }
      else if (# gates in D-frontier == 1)
        unique sensitization;
      else {                  //no. gates == 0
        if (there are untried combinations) {
          set untried combination;
          backtrace_flag = B;
        }
        else
          return (FAIL);
      }
    }
  }
}

final_objective()
{
  mb = 0;
  if (backtrace_flag == A)
    mb = MBack(A);
```

```
  else if (fanout objectives != EMPTY)
    mb = MBack(B);
  if (mb == D) {    //MBack() returns with 'C' or 'D'
    final_objective = FPO;
    return;
  }
  for (;;)
  {
    if (head_objectives == EMPTY)
      mb = MBack(A);
    choose Head Objective;
    if (headline unspecified)
      break;
  }
  Head Objective = Final Objective;
}

MBack(flag)
{
  if (flag == A) {
    backtrace_flag = 0;
    if (# unjustified_lines > 0)
      {initial_objective} = unjustified lines;
    if(fault signal did not reach P.O.)
      add gate in D-frontier to initial objectives;
    {current_objective} = {initial_objective};
    if ({current_objective} != EMPTY) {
      choose current_objective;
      next_obj();
    }
    else {
      if (FPO == EMPTY)
        return(C);
      else
        flag = B;   //force execution of the "flag == B"
                    //code
    }
  }
  if (flag == B) {
    choose FPO p closest to P.O.;
    if ((p reachable from fault line) or ((n0 == 0) or
          (n1 == 0)))
```

```
      next_obj();
    else
      return(D);
  }
}

next_obj()       //next objective
{
  if (current_objective == headline)
    add current_objective to head_objectives;
  else if (current_objective driven by FPO)
    add n0 and n1 to FPO    //(Table 4.1, rule #10);
  else                      //determine next objectives
    backup through gates using Table 4.1 rules #1-9;
//add them to the set of current objectives
}
```

4.8 SOCRATES

FAN started with PODEM and added enhancements whose purpose was to eliminate unnecessary backtracks and reduce the amount of processing time between backtracks. In like manner, Socrates[12] started with FAN and identified enhancements that were able to realize further performance gains. Socrates identified improvements in the implication, unique sensitization, and multiple backtrace procedures. In addition, Socrates added support for complex primitives such as adders, multiplexers, encoders, and decoders, as well as XOR and XNOR gates with an arbitrary number of inputs.

Consider first the implication operation. In Figure 4.15(a) the signal on input A is a 1. That value passes through both OR gates, implying 1s on the outputs of both OR gates, thus implying a 1 on the output of the AND gate. Now consider the situation in Figure 4.15(b). The output of the AND gate is a 0, which implies that input A must be a 0. This follows from the logic identity $(A \Rightarrow D) \Leftrightarrow (\sim D \Rightarrow \sim A)$, known as the contrapositive, where the tilde (~) is used to denote the complement. The value of this observation lies in the fact that if a 0 is assigned to the output of the AND gate during a backtrace, input A must be assigned a 0; it cannot be treated as a decision and postponed until later. This, in turn, can lead to earlier recognition of conflicts and reduce the number of backtracks.

To recognize these situations, a learning phase is performed prior to entering the test generation phase. During this learning phase, a 0 is applied to net n_i and implied. The result is then analyzed. This is repeated using the value 1. Assume that, during the implication, n_i is initialized to the value v_i, $v_i \in \{0,1\}$, and net n_j receives the value v_j, $v_j \in \{0,1\}$ as a result of the implication, that is, $(\text{value}(n_i) = v_i) \Rightarrow (\text{value}(n_j) = v_j)$. Let n_j be driven by gate g. Thus if (1) v_j requires all inputs of g to have noncontrolling

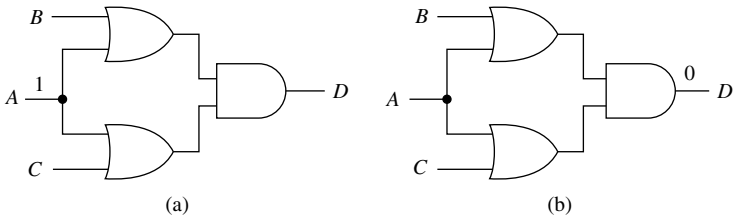


Figure 4.15 Implications.

values and (2) a forward implication has contributed to the assignment v_j to net n_j, then the implication (value(n_j) = $\overline{v}_j$) $\Rightarrow$ (value(n_i) = $\overline{v}_i$) is worth learning. Condition 1 is satisfied if v_j is 1(0) and g is an AND/NOR (OR/NAND) gate, or if g is an XOR or XNOR gate. An additional function, check_path(n_j, n_i), checks the network to ensure that there is no directed path from n_j to n_i. If the circuit is combinational and rank-ordered and if $j > i$, check_path() returns the value 0.This ensures that condition 2 has been satisfied.

It is possible that the procedure just described will not find an implication where an implication exists; that is, the procedure is a sufficient, but not necessary, condition to establish than an implication cannot be performed by the implication procedure. However, the payback from the process, in general, outweighs the cost of performing the learning operation.

The unique sensitization in FAN handles situations in which the D-frontier consists of a single gate and all paths from the D-frontier to the primary output pass through that gate. Like improved implication, the unique sensitization is accomplished by means of circuit preprocessing.

Definition 4.1 A signal y is said to dominate signal x, y $\in$ dom(x), if all directed paths from x to the primary outputs of the circuit pass through y.

Let x be the only signal in the D-frontier. Let the set of signals dom(x) = {y_1, y_2, ..., y_n} be the output signals of their corresponding gates in the set G = {g_1, g_2, ..., g_n}. Then, for all gates $g \in G$, the noncontrolling value is assigned to all those inputs of g that cannot be reached from x on any signal path. This is illustrated in Figure 4.16. The output of gate a has a D assigned. The signal diverges at gates b and c and then reconverges at inputs e and f of gate g. In this situation the signal d must be set to 1, the noncontrolling value.

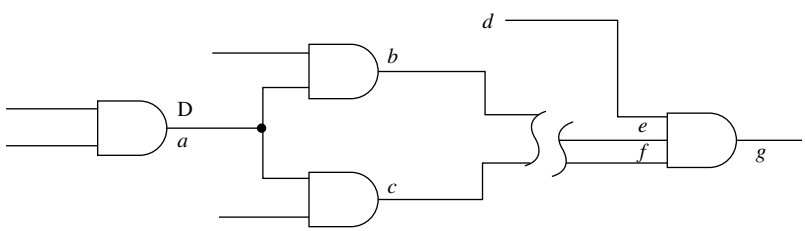


Figure 4.16 Improved unique sensitization.

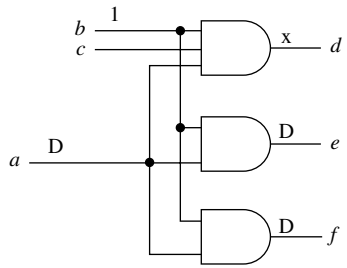


Figure 4.17 Uniquely sensitizing multiple paths.

Definition 4.2 A signal y is said to be the *immediate dominator* of signal x if $y \in \text{dom}(x)$, and y is the element of dom(x), that has the lowest circuit level.

In this definition, the level of an element in a combinational circuit is determined by rank-ordering the circuit elements (cf. Section 2.6). If the immediate dominators of all signals are known, the dominators of any signal x can be determined recursively. For example, if signal y is an immediate dominator signal x, and signal z is an immediate dominator of signal y, then signal z is a dominator of signal x.

An additional rule for unique sensitization is required in order to handle the situation depicted in Figure 4.17. Assume that signal a is the only signal in the D-frontier, or a dominator of the only signal in the D-frontier. It branches out to three AND gates, all of which have an input from signal b. In addition, one of the AND gates has a third input c. Assume signal a is the only signal in the D-frontier, or a dominator of the only signal in the D-frontier, and it branches out to gates g_1, g_2, ..., g_n, all of which require the same noncontrolling value 0 or 1. If signal b branches out to all the same gates g_1, g_2, ..., g_n, then b is assigned the noncontrolling value.

The multiple backtrace in Socrates takes advantage of the fact that some commonly occurring circuit configurations are processed as primitives. For example, the gates M, N, O, and P in Figure 4.1 constitute an XOR. If the diagram is altered so that gates K and L drive an XOR, the circuit function remains unchanged but three fanout branches are eliminated. An important point to bear in mind about the XOR is that a sensitized path on one input of a two-input XOR is propagated to its output regardless of the binary value on the other input. For example, the values (D,0) produce a D on the output, and (D,1) produce a $\overline{\text{D}}$ on the output. Therefore, when propagating through an XOR or XNOR, it is only necessary to ensure that the other input has a known value and that both inputs do not have sensitized values. This line of reasoning can be extended to n-input XOR gates, which Socrates supports.

PODEM was not adversely affected by the XOR because it did not attempt to justify assignments on the inputs of XOR gates—in contrast to the D-algorithm, which, particularly in parity trees, can thrash about trying to find a self-consistent set of assignments to the circuit, making and changing assignments to resolve conflicts. However, representing the XOR as a primitive simplifies test generation because it

TABLE 4.3 Multiple Backtrace for Two-Input XOR

Formula		Condition
$n_0(x_1) = n_0(y)$	$n_0(x_2) = n_0(y)$	$c_{00} < c_{11}$
$n_1(x_1) = n_0(y)$	$n_1(x_2) = n_0(y)$	$c_{00} \geq c_{11}$
$n_0(x_1) = n_0(x_1) + n_1(y)$	$n_1(x_2) = n_1(x_2) + n_1(y)$	$c_{01} < c_{10}$
$n_1(x_1) = n_1(x_1) + n_1(y)$	$n_0(x_2) = n_0(x_2) + n_1(y)$	$c_{01} \geq c_{10}$

can be recognized as such, whereas representing it as a collection of lower-level gates doesn't solve the problem that caused the D-algorithm to thrash about and simply introduces more fanout points, which introduce additional processing. Socrates uses Table 4.3, analogous to Table 4.1, to compute the objective triplets when an XOR is encountered:

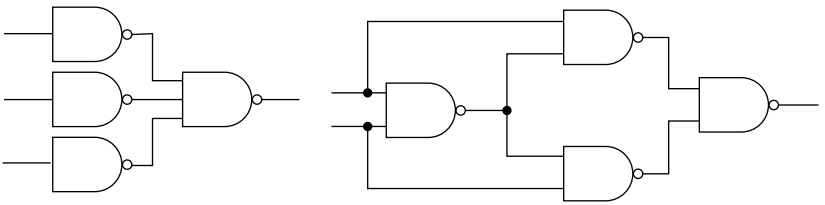
In this table, c_{ij} represents the controllability cost for setting x_1 to i and x_2 to j, for $i, j \in \{0,1\}$, where x_1 and x_2 are the inputs to the two-input XOR and y is the output. Other, higher-level primitives require similar specific formulas. The main advantage of higher-level primitives is the reduction of fanout branches. But it is sometimes possible to realize opportunities not readily inferred from the gate level model. For example, if a two-input multiplexer has 1s on both data inputs, the output is going to be 1, even if the select line has an X.

4.9 THE CRITICAL PATH

The D-algorithm starts at a fault origin and works outward from there, stretching the sensitized path toward outputs and inputs. PODEM selects a fault and attempts to sensitize a path by working from the primary inputs. FAN adopts features from both the D-algorithm and PODEM. The *critical path*[13] starts at primary outputs and works back toward primary inputs. It has been implemented commercially as LASAR (logic automated stimulus and response)[14] and was the ATPG companion to the LASAR deductive fault simulator mentioned in the summary to Chapter 3. It enjoyed considerable commercial success for several years, having been marketed by several companies. Like the simulator, the ATPG only recognizes the NAND gate. This not only simplified deductive fault simulation computations, but also simplified computations for ATPG. In order for critical path to process circuits implemented with other logic primitives, those primitives must be remodeled in terms of the NAND gate (cf. Figure 4.18).

Processing rules for a circuit being processed by critical path are defined in terms of *forcing values* and *critical values* as they apply to the NAND gate. The forcing rules for an n-input NAND gate are as follows:

1. If the output of a NAND gate is 0, then the inputs are all forced to 1.
2. If the inputs are all 1, the output is forced to 0.
3. If the output is 1 and all inputs except input i are 1, then input i is forced to 0.



(a) 3-input OR (b) Exclusive-OR

Figure 4.18 Some simple transformations.

A value on a node is *critical* if its existence is required to establish a test. The rules are as follows:

1. If the output of a NAND gate is a 0, and it is critical, then the inputs are all critical 1s.
2. If the output is a critical 1 and if all inputs except input *i* are 1s, then input *i* is a critical 0.

If a NAND gate has a critical 0 on its input, then the other input assignments are all *necessary* 1s; that is, it is necessary that they be 1s in order for input *i* to be critical. In order for a NAND gate to provide a necessary 1 on its output, at least one of its inputs must have a 0 assigned. That input is always *arbitrary* or noncritical.

The creation of a test starts with the selection of an output pin and assignment of a 0 or 1 state to that pin. From that pin an attempt is made to extend critical values as far back as possible toward the inputs using the rules for establishing critical values. Then, after the path is extended as far back as possible, the necessary states are established. When complete, a critical path extends from an output pin back to either some internal net(s) or to one or more input pins (or both). The critical paths define a series of nets or signal paths along which any gate input or output will, if it fails, cause the selected output to change from a correct to an incorrect value. Since the establishment of a 0 on an output pin requires 1s on all the inputs to the NAND gate connected to that output, it is possible to have several critical paths converging on an output pin.

Upon successful creation of a test, the next test begins by permuting the critical 0 on the lowest-level NAND gate that has one or more inputs not yet tested—that is, the critical 0 closest to the primary inputs. The 0 is assigned to one of the other inputs to that NAND gate and the input that was 0 is now assigned the value 1. The test process then backs up again from the critical 0 to primary inputs, attempting to satisfy these new assignments. A successful test at any level may result in a critical 0 at a lower level becoming a candidate for permutation before another critical 0 on the NAND gate that was just processed. However, once selected, a NAND gate will be completely processed before another one is selected closer to the output. Eventually, after all the inputs to the gate driving the output have been permuted, the output pin is then complemented, if the complement value hasn't already been processed, and the process is repeated.

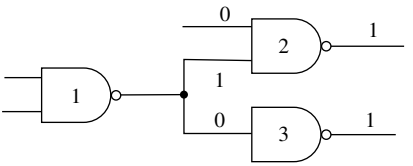


Figure 4.19 Critical assignments.

The practice of postponing necessary assignments until the critical path(s) have been extended as far back as possible can help to minimize the number of conflicts that occur. Figure 4.19 illustrates a situation where a net fans out to two NAND gates (gate 3 is actually an inverter). Assuming that the outputs of gates 2 and 3 are both critical, if the upper input of gate 2 is established as far back as possible, and the necessary 1 on the lower input to gate 2 is extended, the assignments on gate 2 will later have to be reversed in order to get a 0 on the input to gate 3. Since the 1 on the output of gate 3 is critical, by the rules for critical assignments, the input to gate 3 is also critical; hence it will be processed before the necessary 1 on the input to gate 2. This avoids having to undo some assignments.

Conflicts can occur despite postponement of necessary assignments. When this occurs, the rule is to permute the lowest arbitrary assignment that will affect the conflict. This is continued until a self-consistent set of assignments is achieved. These concepts will be illustrated using the circuit of Figure 4.20.

Example The first step is to assign a 0 to the output *F*, which implies 1s on all the inputs to gate number 8. Then gate 5 is selected in an attempt to extend the critical path through one of its inputs. That requires inputs 1, 2, 3 = (0,1,1). Hence, input 1 is critical and inputs 2 and 3 are necessary. We must then get a 1 on the output of gate 6. We try to extend another critical path. Since the middle input of gate 6 is the complement of the value on input 3, a second critical path cannot be extended back through gate 6 without disturbing the critical path already set up through gate 5. However, the values already assigned on 1,2, and 3 do satisfy the critical 1 value needed at the output of gate 6.

We then try to extend the critical path through gate 7. This also fails. Worse, still, the values already assigned to the inputs are in conflict with the critical 1 assigned to

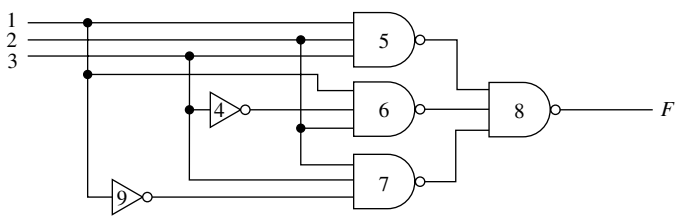


Figure 4.20 Creating a critical path.

the output of gate 7 because they force gate 7 to produce a logic 0. We go back to gate 5 and permute the assignments on its inputs. A critical 0 is assigned to the middle input and we now have an assignment (1, 2, 3) = (1, 0, 1) that produces 1s on the outputs of 5, 6, and 7. A critical path now exists from input 2, through gates 5 and 8, to the output *F*. Critical paths also exist from the outputs of gates 6 and 7 to the output *F*. ■■

4.10 CRITICAL PATH TRACING

The purpose of critical path tracing (CPT) is to estimate the fault coverage provided by a test program.[15,16] CPT performs a logic simulation on a circuit and then, based on simulation results, it identifies gates with sensitive values, where gate input *i* is *sensitive* if complementing the value of *i* changes the value of the gate output. Sensitive inputs can be identified on the basis of the *dominant logic value (DLV)*. A DLV at a gate input is one that forces an output to a value, regardless of the values on the other inputs. For example, the DLV for AND and NAND gates is 0, while the DLV for OR and NOR gates is 1. Note that, unlike the previous subsection where critical path ATPG required all gates to be NANDs, CPT recognizes critical values for ORs, NORs, and ANDs, in addition to NANDs. The following statements hold for DLVs:

1. If only one input *i* has a DLV, then *i* is sensitive.
2. If all inputs have the complement of the DLV, then all inputs are sensitive.
3. If neither 1 or 2 holds, then no input is sensitive.

A net *n* is said to have *critical value* $v \in \{0,1\}$ in a test *T* if *T* detects the fault *n* SA*v*. CPT involves tracing from POs, which are critical (assuming they have a known value) and backtracing along sensitive paths to create critical paths. The critical paths identify detectable faults. In the circuit in Figure 4.21 the dots denote inputs that are sensitive. The bold lines indicate a critical path. At gate *G*, both of the inputs are DLVs, so neither of them is sensitive and the backtrace stops there. Faults along the critical path can all be declared detected.

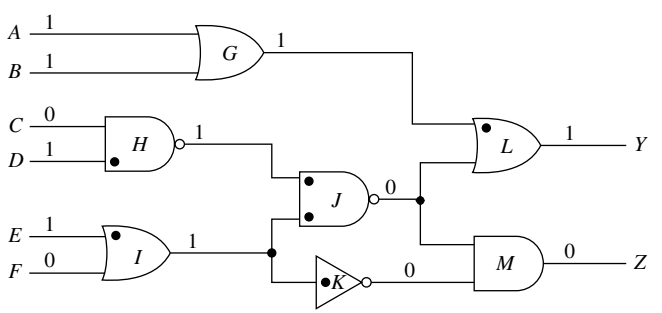


Figure 4.21 Tracing the critical path.

Ignore for the moment the output *Y* and consider just the cone feeding output *Z*. At gate *M* both input values are DLVs, so neither input is sensitive. But inspection of the circuit suggests that an SA0 on the stem emanating from gate *I* is detectable at output *Z*. A concurrent fault simulation of the circuit would show that if the stem were SA0, then the outputs of both *J* and *K* would be 1; hence the output of *M* would be 1 in the presence of the fault and would be detected. Interestingly, if logic simulation produced a 0 on the output of *I*, then both inputs to *M* would be 1; that is, both inputs would have DLVs, and CPT would detect the fault.

CPT preprocesses a circuit to identify its cones, which are then represented as an interconnection of FFRs. After a logic simulation has been performed and sensitive inputs have been marked, CPT backtraces, from a primary output. As it backtraces, it identifies critical paths inside fanout-free regions (FFRs) contained in the cone, where an FFR is a cone (cf. Section 3.6.2) that has no reconvergent fanout. The inputs to a FFR are *fanout branches* (*FOB*) and primary inputs without FOBs. If a stem is encountered during backtrace through a FFR, it is checked to determine if it is critical. If it is critical, then critical path tracing continues from that stem.

If circuits did not contain reconvergent fanout, CPT would be straightforward. However, reconvergence is an attribute of just about all digital circuits, and one of the consequences of reconvergence is *self-masking*, in which a fault effect (FE) propagates along two or more paths and reconverges with opposite parities or polarities at a gate, where the FEs cancel out. As an example, if gate *K* in Figure 4.21 were a buffer, rather than an inverter, then the lower input to *M* would be sensitized. A fault at the stem emanating from *I* could reach the sensitized input through the buffer, but an inverted version would reach the upper input by way of gate *J*. Because of self-masking, stem processing requires a great deal of analysis, and determining criticality of a stem takes up a major part of the computation time for CPT.

One approach to stem processing is to use fault simulation. However, just the stem faults are fault-simulated.[17] If a stem fault is marked as detected, the corresponding FFR is analyzed by backtracing as was described here. Since the number of stem faults is significantly less, often one-third to one-quarter of the total number of faults, the amount of fault simulation time should be significantly reduced, and backtracing the FFRs can be considerably faster than fault simulation for faults in the FFR. However, an unpublished study of concurrent fault simulation for stem faults suggests that even though there are many fewer faults, the amount of CPU time for stem fault simulation can take longer than fault simulation of an industry standard fault list.[18] This is probably due to the fact that two faults are attached to every stem, and one or the other of these two faults propagates on every vector, and it propagates along two or more FOBs, thus generating a large number of fault events.

For CPT, then, the problem is to determine if a stem *S* is critical, given that one or more of its FOBs is critical. The stem *S* was reached from one or more critical FOBs during backtracing. So it would be expected that the stem fault would propagate forward along the critical FOB(s), to the output of the FFR, unless self-masking occurred.

We now provide an overview of stem analysis, but, first, some definitions are in order. The *level* of a net is computed as follows: A primary input is assigned level 0, and the level of a gate output is $1 + i_{max}$ where i_{max} is the highest level among the levels of the gate inputs. If a test T activates fault f in a single-output circuit and if T sensitizes net y to f, but does not sensitize any other net with the same level as y, the y is said to be a *capture line* of f in test T. A test T detects a fault f iff all the capture lines of f in T are critical in T.

In a single-output circuit, a net y that lies on all paths between net x and the PO is said to be a *cover line* of x. If all paths between x and its cover line y have the same inversion parity, then y is said to be an *equal parity cover line* of x. Note that a capture line is defined on the basis of the applied test, while a cover line of x is always a capture line of a fault on x in any test that detects it. Note also that self-masking cannot occur in a region between a stem and its equal parity cover line, hence a stem that has an equal parity cover line is critical in any test in which any of its FOBs is critical. When backtracing, any such stem can be marked as critical.

Some additional properties of FFRs prove to be useful: given a set of inputs $\{x_i\}$ to a FFR, let v_i be the value of x_i for test T and let p_i be the inversion parity of the path from x_i to the FFR output. Then:

1. If fault effects arrive on a subset $\{x_k\}$ of FFR inputs such that at least one input in $\{x_k\}$ is critical and all the inputs in $\{x_k\}$ have the same XOR $p_k \oplus v_k$, then the FFR propagates fault effects.
2. All critical inputs $\{x_j\}$ of an FFR have the same XOR $p_k \oplus v_k$.
3. If FEs arrive only on critical inputs of an FFR, then the FFR propagates FEs.
4. If a fault only affects one FFR input, and that input is noncritical, then the FFR does not propagate the fault effect.

The value of these properties lies in the fact that they can lead to efficient stem analysis by obviating the need to analyze the gates inside a FFR. If a property holds, then a decision can be immediately made as to whether a fault propagates to the output of the FFR.

As pointed out earlier, the analysis can sometimes miss faults that actually are detected. Hence, CPT can turn out to be slightly pessimistic. It is argued that this approximation is not serious since the situation rarely occurs and, additionally, the stuck-at fault model is, itself, only an approximation.

4.11 BOOLEAN DIFFERENCES

Up to this point the methods that have been described can be characterized as path tracing. A netlist is provided and the algorithm or procedure attempts to create a sensitized path from the fault to an output pin. We now turn our attention to Boolean differences. In this method, an equation describes the set of tests for a given fault. The equation is usually quite complex, and a large part of the work involves reducing the equation to a manageable size.

Given a function F that describes the behavior of a digital circuit, if a fault occurs that transforms the circuit into another circuit whose behavior is expressed by F^*, then the 1-points of the function T,

$$T = F \oplus F^*$$

define the complete set of tests capable of distinguishing between F and F^*.

Example A test will be created for a shorted inverter (gate 5) in the circuit of Figure 4.22. The equation for circuit behavior is

$$F = x_4 \cdot (\overline{x_1} + x_2) \cdot (x_1 + x_3)$$

With a shorted inverter, the equation becomes

$$F^* = x_4 \cdot (x_1 + x_2) \cdot (x_1 + x_3)$$

Then

$$\begin{aligned} F \oplus F^* &= \overline{F} \cdot F^* + F \cdot \overline{F^*} \\ &= \overline{x_2} \cdot x_4 \cdot (x_1 + x_3) \end{aligned}$$

It can be seen from this equation that if $x_2 = 0$ and $x_4 = 1$, then a 1 on either x_1 or x_3 will cause the fault-free circuit and the faulted circuit to produce different outputs (verify this); hence a test has been found that is capable of detecting the presence of the shorted inverter. ■■

For the moment we restrict our attention to input faults. Given a function $F(x_1,x_2,...,x_n)$, the Boolean difference[19] of F with respect to its ith input variable is defined as

$$D_i(F) = F(x_1, \ldots, x_i, \ldots, x_n) \oplus F(x_1, \ldots, \overline{x_i}, \ldots, x_n)$$

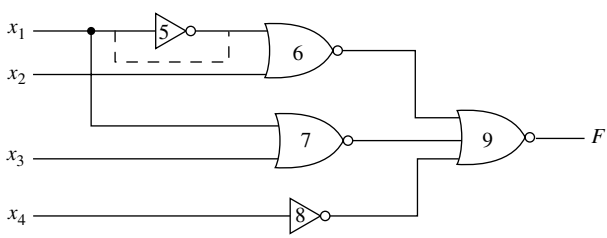


Figure 4.22 Circuit with shorted inverter.

The following properties[20] hold for the difference operator (in what follows, the AND operation takes precedence over the exclusive-OR):

1. $D_i(F) = D_i(\overline{F})$
2. $D_i(F(x_1,...,x_i,...,x_n)) = D_i(F(x_1,...,\overline{x}_i,...,x_n))$
3. $D_i(D_j(F)) = D_j(D_i(F))$
4. $D_i(F \cdot G) = F \cdot D_i(G) \oplus G \cdot D_i(F) \oplus D_i(F) \cdot D_i(G)$
5. $D_i(F + G) = \overline{F} \cdot D_i(G) \oplus \overline{G} \cdot D_i(F) \oplus D_i(F) \cdot D_i(G)$
6. $D_i(F \oplus G) = D_i(F) \oplus D_i(G)$

We outline the proof for property 4, but first we state some properties of the Exclusive-OR operator:

(a) $F \oplus F = 0$
(b) $F \oplus 0 = F$
(c) $F \oplus G = G \oplus F$
(d) $G = F \oplus F \oplus G$
(e) $F + G = F \oplus G \oplus F \cdot G$
(f) $F \cdot G \oplus F \cdot H = F \cdot (G \oplus H)$
(g) $F(x) = x_i \cdot F(x_1, ..., 1, ..., x_n) \oplus \overline{x}_i \cdot F(x_1, .., 0, ..., x_n)$

We now sketch the proof. For notational convenience we omit the subscript associated with the variable x_i and the functions F and G. It is understood that the functions are differenced with respect to the ith variable, x_i, and that F_e, $e \in \{0,1\}$, denotes $F(x_1, ... , e, ... , x_n)$. The property (g) will be used to expand the left-hand side:

$$\begin{aligned} D_i(F \cdot G) &= D_i[x \cdot F_i \oplus \overline{x} \cdot F_0) \cdot (x \cdot G_1 \oplus \overline{x} \cdot G_0)] \\ &= [(x \cdot F_1 \oplus \overline{x} \cdot F_0) \cdot (x \cdot G_1 \oplus \overline{x} \cdot G_0)] \\ &\quad \oplus [(\overline{x} \cdot F_1 \oplus x \cdot F_0) \cdot (\overline{x} \cdot G_1 \oplus x \cdot G_0)] \\ &= x \cdot F_1 \cdot G_1 \oplus \overline{x} \cdot F_0 \cdot G_0 \oplus \overline{x} \cdot F_1 \cdot G_1 \oplus x \cdot F_0 \cdot G_0 \end{aligned}$$

We take note of the first two terms in the expansion and use properties (a) and (b) to add the terms indicated in braces:

$$\begin{aligned} &= G_1 \cdot x \cdot F_1 \oplus \{G_1 \cdot \overline{x} \cdot F_0 \oplus G_1 \cdot x \cdot F_0\} \\ &\quad \oplus G_0 \cdot \overline{x} \cdot F_0 \oplus \{G_0 \cdot x \cdot F_1 \oplus G_0 \cdot x \cdot F_1\} \\ &\quad \oplus \overline{x} \cdot F_1 \cdot G_1 \oplus x \cdot F_0 \cdot G_0 \end{aligned}$$

The braces are dropped, terms 1 and 2 are grouped, as are 4 and 5, and properties (c) and (f) are used, thereby yielding

$$= \{[(x \cdot F_1 \oplus \bar{x} \cdot F_0) \cdot G_1] \oplus [(x \cdot F_1 \oplus \bar{x} \cdot F_0) \cdot G_0]\}$$
$$\oplus \bar{x} \cdot F_0 \cdot G_1 \oplus x \cdot F_1 \cdot G_0 \oplus \bar{x} \cdot F_1 \cdot G_1 \oplus x \cdot F_0 \cdot G_0$$

The term in braces is recognized as $F \cdot D_i(G)$. This yields

$$D_i(F \cdot G) = F \cdot D_i(G) \oplus x \cdot G_0 \cdot D_i(F) \oplus \bar{x} \cdot G_1 \cdot D_i(F)$$

where the second and third terms were obtained by grouping product terms with a common x or $\bar{x}$ variable and factoring. Factoring once again yields

$$D_i(F \cdot G) = F \cdot D_i(G) \oplus D_i(F) \cdot [\bar{x} \cdot G_1 \oplus x \cdot G_0]$$
$$= F \cdot D_i(G) \oplus D_i(F) \cdot [G \oplus G \oplus \bar{x} \cdot G_1 \oplus x \cdot G_0]$$
$$= F \cdot D_i(G) \oplus \mathrm{G} \cdot D_i(F) \oplus D_i(F) \cdot [G \oplus \bar{x} \cdot G_1 \oplus x \cdot G_0]$$

When G is expanded to $x \cdot G_1 \oplus \bar{x} \cdot G_0$, the expression in square brackets is recognized as $D_i(G)$. We leave the details as an exercise.

Now consider again the circuit of Figure 4.22. We will attempt to create a test for input x_3 SA0. However, rather than try to solve the problem by brute force as we did previously, this time we attempt to exploit the six relationships that we have just defined. We start by defining the following functions:

$$g = x_4$$
$$h = (\bar{x}_1 + x_2)(x_1 + x_3)$$

Property 4 can now be used to compute the difference relative to input x_3:

$$D_3(g \cdot h) = g \cdot D_3(h) \oplus h \cdot D_3(g) \oplus D_3(g) \cdot D_3(h)$$

A cursory glance at the expression tells us that much remains to be done. Are there any shortcuts? Fortunately, the answer is yes. We digress briefly to define the concept of independence. A function $F(X)$, $X = (x_1,..., x_i,..., x_n)$, is *independent* of x_i if $F(X)$ is logically invariant under complementation of x_i. This definition leads to:

Theorem 4.1 The function $F(X)$ is independent of x_i iff $D_i(F) = 0$.

If the function $F(X)$ is independent of x_i, then the difference operator possesses the following properties:

7. $D_i(F) = 0$
8. $D_i(F \cdot G) = F \cdot D_i(G)$
9. $D_i(F + G) = \overline{F} \cdot D_i(G)$

Alternatively, if $F(X)$ is a function only of x_i, then

10. $D_i(F) = 1$

With these additional properties, we now return to the problem. Since $g = x_4$ is independent of x_3, it follows that $D_3(g) = 0$; hence

$$D_3(g \cdot h) = g \cdot D_3(h)$$

If two new functions are defined,

$$u = \bar{x}_1 + x_2$$
$$v = x_1 + x_3$$

then property 4 can be applied to $D_3(h)$ to get

$$\begin{aligned} g \cdot D_3(h) &= g \cdot D_3(u \cdot v) \\ &= g \cdot u \cdot D_3(v) \qquad \text{(from property 9)} \end{aligned}$$

Property 5 can now be used to yield

$$D_3(x_1 + x_3) = \bar{x}_1 \cdot D_3(x_3) \oplus \bar{x}_3 \cdot D_3(x_1) \oplus D_3(x_3) \cdot D_3(x_1)$$

The independence theorem permits the last two terms to be discarded, yielding

$$\begin{aligned} D_3(F) &= x_4 \cdot (\bar{x}_1 + x_2) \cdot \bar{x}_1 \cdot D_3(x_3) \\ &= \bar{x}_1 \cdot x_4 (\bar{x}_1 + x_2) \\ &= \bar{x}_1 \cdot x_4 \end{aligned}$$

The circuit of Figure 4.22 is a multiplexer with an enable input. The select line is x_1, the enable is x_4, and the data inputs are x_2 and x_3. The final equation says that an error on input x_3 will be visible at the output if the multiplexer is enabled and if input x_3 is selected, $(x_1 = 0)$. The Boolean difference method has, in effect, created a sensitized path from input x_3 to an output. It now remains but to apply a 1 and a 0 to x_3 in order to exercise and completely test the path from x_3 to the output.

Up to this point the discussion has been limited to primary inputs. It is also possible to detect faults internal to a circuit using the Boolean difference. First, consider the internal node to be just another input x_{n+1}. Then express the behavior of the circuit as a function of the original inputs and the new input. The internal node will, in general, be some function G of the same set of inputs. To test for a SA1 (SA0), create a path from the newly created "input" to the output and, in addition, force that "input" to assume the value 0(1). Hence, we want to compute the solution for

$$\bar{x}_{n+1} \cdot D_{n+1}(F) = 1 \qquad \text{for a SA1 fault}$$
$$x_{n+1} \cdot D_{n+1}(F) = 1 \qquad \text{for a SA0 fault}$$

Example In order to contrast the amount of computation required, we will again create a test for the shorted inverter, this time using the Boolean difference. The output of gate 5 is now treated as an input. F is expressed as

$$F = x_4 \cdot (x_2 + x_5) \cdot (x_1 + x_3)$$

In this case, the function G is simply x_1.

Now applying the difference operator and the given properties to F yields

$$\begin{aligned} G \cdot D_{n+1}(F) &= G \cdot [x_4 \cdot (x_1 + x_3) \cdot D_5(x_2 + x_5)] \qquad \text{(properties 4 and 7)} \\ &= G \cdot [x_4 \cdot (x_1 + x_3) \cdot (x_2 \cdot D_5(x_5))] \qquad \text{(property 5)} \\ &= G \cdot [x_4 \cdot (x_1 + x_3) \cdot x_2] \qquad \text{(property 10)} \end{aligned}$$

The expression within the square brackets specifies the necessary conditions on the inputs in order to propagate the fault to the output. Since the fault is a shorted inverter, either value of x_1 will distinguish the faulty circuit from the fault-free circuit. ■■

The Boolean differences have been developed quite thoroughly; for instance, if G is a function $G(u,v)$ of u and v, and $u = u(x_1, ..., x_n)$, $v = v(x_{n+1}, ..., x_{n+m})$, where u and v share no variables in common, then the following *chain rule* holds:

$$D_i(G) = D_1(G) \cdot D_i(u)$$

where $D_1(G)$ is the difference of G with respect to u and $D_i(u)$ is the difference of u with respect to its ith variable. With the chain rule, the Boolean differences behaves much like the path sensitization approaches.

Example The chain rule will be applied to input x_3 of the circuit of Figure 4.22. The first step is to separate the expression for the circuit into subexpressions that have no variables in common:

$$F = \overline{\bar{x}_2 \cdot x_1 + \bar{x}_1 \cdot \bar{x}_3 + \bar{x}_4}$$

if

$$u = \bar{x}_1 \cdot \bar{x}_3$$
$$v = \bar{x}_2 \cdot x_1 + \bar{x}_4$$

then

$$F = \overline{u + v}$$

and

$$D_3(F) = D_1(F) \cdot D_3(u)$$

From this point it is a simple exercise to compute the final result, which is left as an exercise. ■■

4.12 BOOLEAN SATISFIABILITY

The Boolean satisfiability algorithm is an ATPG method for combinational circuits that is not purely structural nor purely algebraic.[21] It creates a formula expressing the Boolean difference between the good and faulted circuits, then it applies a Boolean satisfiability algorithm to the resulting formula. The satisfiability algorithm derives a *conjunctive normal form (CNF)* description of the circuit from the netlist. Like Boolean difference the good and faulty circuit descriptions are XOR'ed. The algorithm then attempts to find a minimal solution for the XOR'ed circuit.

Consider the equation $Z = X$. In terms of logic, this equation is equivalent to $(Z \rightarrow X) \cdot (X \rightarrow Z)$. We now use another logic identity. In propositional logic, the expression $(Z \rightarrow X)$ is equivalent to $(\overline{Z} + X)$; that is, a false premise can imply anything. The expression $(Z \rightarrow X) \cdot (X \rightarrow Z)$ now becomes $(\overline{Z} + X) \cdot (Z + \overline{X})$. For this expression to be true, either X and Z must both be true (1), or both must be false (0).

We now take the discussion a step further by means of the equation $Z = X \cdot Y$, for the AND gate. This equation leads to the following formula: $(Z \rightarrow X \cdot Y) \cdot (X \cdot Y \rightarrow Z)$. The next step yields

$$(\overline{Z} + X \cdot Y) \cdot (\overline{X} \cdot \overline{Y} + Z) = (\overline{Z} + X) \cdot (\overline{Z} + Y) \cdot (\overline{X} + \overline{Y} + Z).$$

The individual terms are referred to as clauses. Clauses with one, two, or three terms are unary, binary, or ternary clauses, respectively. For any two-input AND gate the expression evaluates to 1 only if the values are consistent with the values in the truth table. Table 4.4 lists formulas for several gate types. Formulas for logic gates with three or more inputs can be deduced from the table and the preceding discussion.

TABLE 4.4 Formulas for Satisfiability

Formula	Gate Type
$(\overline{Z} + X) \cdot (Z + \overline{X})$	Buffer
$(Z + X) \cdot (\overline{Z} + \overline{X})$	Inverter
$(\overline{Z} + X) \cdot (\overline{Z} + Y) \cdot (\overline{X} + \overline{Y} + Z)$	Two-input AND
$(Z + X) \cdot (Z + Y) \cdot (\overline{X} + \overline{Y} + \overline{Z})$	Two-input NAND
$(Z + \overline{X}) \cdot (Z + \overline{Y}) \cdot (X + Y + \overline{Z})$	Two-input OR
$(\overline{Z} + \overline{X}) \cdot (\overline{Z} + \overline{Y}) \cdot (X + Y + Z)$	Two-input NOR
$(\overline{X} + Y + Z) \cdot (X + \overline{Y} + Z) \cdot (X + Y + \overline{Z}) \cdot (\overline{X} + \overline{Y} + \overline{Z})$	Two-input XOR

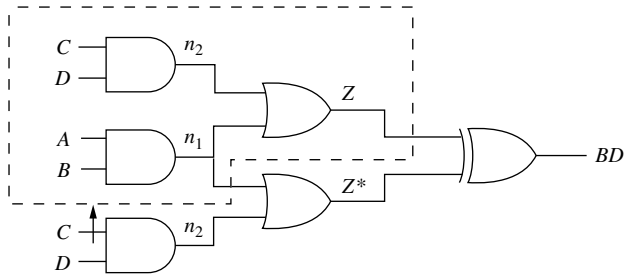


Figure 4.23 Circuit for satisfiability calculations.

Given the circuit in Figure 4.23, the original circuit $Z = A \cdot B + C \cdot D$ is indicated by the dashed lines. It can be described in conjunctive normal form by means of the following formula:

$$(\overline{n}_1 + A) \cdot (\overline{n}_1 + B) \cdot (\overline{A} + \overline{B} + n_1) \cdot (\overline{n}_2 + C) \cdot (\overline{n}_2 + D) \cdot (\overline{C} + \overline{D} + n_2)$$
$$\cdot (Z + \overline{n}_1) \cdot (Z + \overline{n}_2) \cdot (n_1 + n_2 + \overline{Z})$$

We hypothesize an SA1 fault on input C. Then, as in the Boolean difference, we take the XOR of the fault-free and faulty circuits. The operation is combined in Figure 4.23 where $BD = Z \oplus Z^*$. Note that the two circuits, Z and Z*, share a common subcircuit, the AND gate with inputs A and B. The CNF formula for this subcircuit becomes

$$(\overline{n}_1 + A) \cdot (\overline{n}_1 + B) \cdot (\overline{A} + \overline{B} + n_1) \cdot (\overline{n}_2 + C) \cdot (\overline{n}_2 + D) \cdot (\overline{C} + \overline{D} + n_2)$$
$$\cdot (Z + \overline{n}_1) \cdot (Z + \overline{n}_2) \cdot (n_1 + n_2 + \overline{Z})$$
$$\cdot (\overline{n}_2' + C') \cdot (\overline{n}_2' + D) \cdot (\overline{C}\text{`} + \overline{D} + n_2') \cdot (C') \cdot (Z^* + \overline{n}_1) \cdot (Z^* + \overline{n}_2') \cdot (n_1 + n_2' + \overline{Z}^*)$$
$$\cdot (\overline{Z} + Z^* + BD) \cdot (Z + \overline{Z}^* + BD) \cdot (\overline{Z} + \overline{Z}^* + \overline{BD}) \cdot (Z + Z^* + \overline{BD})$$

In this formula the first two lines correspond to the fault-free circuit enclosed in the dashed lines. The third line corresponds to the path back from Z^* to the inputs. Because the AND operation is idempotent, it is not necessary to repeat the AND gate driving n_1. Furthermore, we have imposed an additional requirement. Since we are testing for a SA1 on input C', we add the term (C') on line 3, which can only be true if C' is 1. The fourth line in this formula represents the XOR.

This represents a rather prodigious formula for such a small circuit. A solution to this formula is a set of binary values for the variables that cause the formula to evaluate to 1. To find a solution, note that two-input AND/OR gates contribute two binary clauses and one ternary clause. The binary clauses will be referred to as 2CNF clauses. Note also that if a circuit is made up entirely of gates that have two inputs, then 66.6% of the clauses will be in 2CNF. In practice, it is more likely that

80% to 90% of the clauses will belong to 2CNF. This observation suggests the following approach to finding a consistent set of assignments:

- Assign values to members of 2CNF in some methodical way.
- Use the ternary (and other) clauses as constraints.

We begin by defining an array V of 2CNF variables. A pointer i points to the first unbound variable in V, it is initialized to 0. The variable *dir* is used to keep track of whether we are proceeding forward or backtracking, it is initialized to indicate forward processing. During processing, $i > 0$, the sequence of bound values $V[0]$, $V[1]$, ..., $V[i-1]$ represents the *current prefix* of V. The goal is to find a set of assignments to the variables in V that is consistent with the ternary clauses. It is also advantageous to find inconsistencies as quickly as possible. For example, variable P may appear in five binary clauses, and variable Q may appear in two binary clauses. In general, conflicts are more likely to be found if P is assigned before Q.

Other strategies to reduce the amount of calculations include assigning and implying unary clauses, as well as other variables that have known values. For example, in the example above, with a SA1 on input C, the PDCF is C, $D = (0,1)$. Also, BD must equal 1; else we do not have a test. These assignments can be immediately implied. They in turn imply other assignments, with the result that we are left with the binary clause $(\overline{A + B})$. Either A or B can be assigned a 0 to force this binary clause to be 1.

Boolean satisfiability can also benefit from strategies like those used by FAN and Socrates. If it is known that a fault must propagate through an AND gate or an OR gate, then the other inputs to that gate must be set to noncontrolling values. The learned implications of Socrates can also contribute to improvements in performance. The satisfiability algorithm is described below in pseudo C code.

```
SAT()
{
  dir = 0;  //forward
  V = NULL; //initially, all unbound
  i = 0;    //point to V(0), the first unbound variable
  for(;;) {
    if(dir == Forward) {
      for(; i < size(V); i = i+1)   //find unbound entry
        if (V[i] is bound)
          break;
      if(i == size(V))
        return (SUCCESS);
      V[i-1] = 0;
      set implications of V[i-1];
      i = i + 1;
    }
```

```
    else {     //dir == Backward
      if (i == 0)
        return (FAIL);
      temp = V[i-1];
      undo implications of V[i-1];
      set V[i-1] unbound;
      if(temp == 0) {
        V[i-1] = 1;
        set implications of V[i-1];
      }
      else
        i = i-1;
    }
    if(no clause falsified)
      dir = Forward;
    else
      dir = Backward;
  }
}
```

4.13 USING BDDs FOR ATPG

Boolean difference can find a test for a fault if that fault is detectable. A combinational network is compared (exclusive-ORed) against a faulted version of that same network, and the solution is an equation describing the entire solution space for the fault. Because of its general nature, Boolean difference can be applied to any faulted network, not just a network with an SA1 or SA0. Boolean satisfiability provides a method for creating formulas describing fault-free and faulted circuits, and it provides a method for solving the formulas. The method we now present also solves the problem of exclusive-ORing a fault-free and a faulty circuit. The use of binary decision diagrams (BDDs) parallels that of Boolean difference. Given a reduced, ordered BDD (ROBDD) for a fault-free network, along with an ROBDD for the faulted network, the XOR of these two ROBDDs produces a BDD that describes the entire solution space for the fault. Unlike path tracing methods, the amount of time required to create a solution is independent of whether or not a solution exists. We will look at an example in which a test for a stuck-at fault is generated using ROBDDs. That will be followed by a look at research into generating fault lists based on BDDs.

4.13.1 The BDD XOR Operation

Section 2.11 presented a discussion of binary decision diagrams (BDDs). During that discussion some algorithms were presented, including the Traverse, Reduce,

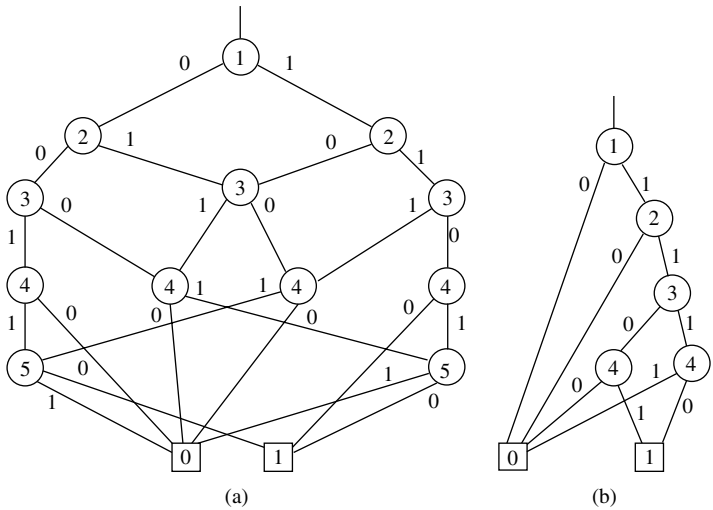


Figure 4.24 ROBDD for SA0 on gate *K*.

and Apply. Section 2.11.3 presented an example in which a BDD for a circuit was constructed from BDDs for two subcircuits. The subsequent BDD was then reduced. This can be continued incrementally until an entire netlist is represented by a ROBDD.

In Section 2.12 a ROBDD was presented corresponding to the netlist in Figure 4.1 (originally Figure 2.43). Here we present, in Figure 4.24(a), an OBDD (not reduced) for Figure 4.1, but with a stuck-at fault on input 3 of gate *K*. There are two differences between this BDD and the BDD in Section 2.12. First, the 0-edge and 1-edge from vertex 5, reached by traversing edges 1, 1, 0, 1, has 0- and 1-edges terminating at terminal vertices 1 and 0, respectively, whereas in the BDD representing the unfaulted circuit, the 0- and 1-edge from vertex 5 terminate at terminal vertices 0 and 1, respectively. The second difference occurs in vertex 4, reached by traversing edges 1, 1, 1. In the original BDD the 0-edge from that vertex terminates on terminal vertex 1; in the BDD representing the faulted circuit, the 0-edge terminates on terminal vertex 0.

The ROBDD shown in Figure 4.24(b) is the result of using Apply to compute the XOR of the ROBDD in Figure 2.45 and the OBDD in Figure 4.24(a). The closed form Boolean expression for this graph is $I_1 \cdot I_2 \cdot (I_3 + I_4)$. Although that expression represents the entire realm of solutions for the stuck-at fault of input 3 of *K*, for some of the solutions I_5 must be assigned a known value, either 0 or 1, it cannot be left at X.

4.13.2 Faulting the BDD Graph

BDDs can be used to generate test vectors directly for digital circuits—that is, without resorting to the use of a gate-level network. For circuits with a small number of

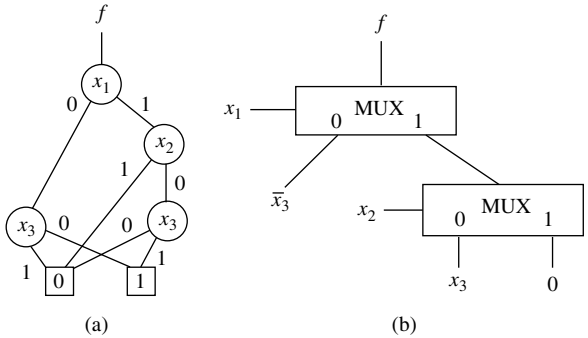


Figure 4.25 BDD implemented with 2-to-1 multiplexers.

inputs, such as the circuit represented by the BDD in Figure 4.25(a), with inputs x_1, x_2, and x_3, an obvious way to generate input vectors is to activate all paths through the diagram. For Figure 4.25(a), the set of vectors would be $x_1,x_2,x_3 = \{0X0,0X1,100,101,11X\}$. If the circuit is implemented using 2-to-1 multiplexers, then stuck-at faults on the inputs of the multiplexers will all be detected. This can be seen in Figure 4.25(b), which implements the BDD in Figure 4.25(a). The set of five vectors that were just computed will detect stuck-at faults on all the I/O pins of these multiplexers. Unfortunately, because of reconvergent fanout inside the multiplexers, it cannot be certain that all the faults inside the multiplexers will be detected.

The use of BDDs to generate test vectors has been studied in some detail. Abadir and Reghbati[22] defined a 2901 4-bit microprocessor slice[23] in terms of BDDs. The individual functions of the device, including the registers, the source and destination selectors, and the ALU, were each modeled using BDDs. Faults were then defined in terms of the signals that connected these functional elements. Two classes of faults were defined: *Class 1 faults* affected the connection variables, and *Class 2 faults* included any functional faults that altered an output of a module while executing one of the module's experiments, where an *experiment* in this context is a path from the output variable to an exit value, and the *exit value* is defined as the value of the terminal vertex. Complete tests for the circuit were based on tests for the individual functions.

Testing for Class 1 faults consisted of assigning values to variables that sensitize a selected input. A test for input *Cin* SA0 in the 4-bit ripple carry adder of Figure 4.26 can be obtained by setting *Cin* = 1 and observing S_0. The response at S_0 will depend on the value of E_0, which in turn depends on A_0 and B_0. However, if it is desired to propagate the SA0 on *Cin* through output S_1, then E_0 must be set to a 1. Testing for Class 2 faults involves walking through all the paths in the BDDs so that all functional possibilities defined by the BDDs are exercised.

In a subsequent study of the effectiveness of test programs based on BDDs, it was pointed out that simply traversing BDDs, using the Class 1 and Class 2 fault models, does not ensure good fault coverage.[24] Traversing BDDs verifies that a device performs

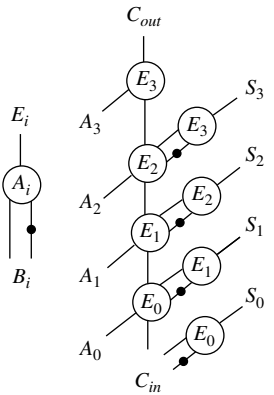


Figure 4.26 BDD for ripple-carry adder.

its intended function, but does not confirm that the device gets the right answer for the right reason—that is, that it does not perform other undesired functions in addition to the intended function. Consider, for example, a four-input AND gate that requires four input events to be true in order to trigger an output event. The negation of any single input event can block the output event from occurring. If two input events are blocking the output event, a logic 0 appears at the output of the AND gate, but it does not confirm that the input event being tested is the one that blocked the output event. Similarly, for an OR gate, any input may trigger an output event, but if two or more inputs are true, no judgment can be made as to whether the input being tested is the one that triggered the output event.

The authors proposed a new functional fault model based on BDDs, and they applied fault simulation to a gate-level model of their circuits to validate the tests that were created. First, they define a functional fault as one that can alter the path of an experiment, but which cannot cause the creation or deletion of vertices, or change vertex connections in the BDD. Then, the following lemma is posited:

Lemma 4.2 For any detectable fault, there always exists a complete path in a BDD that leads to a different exit value.

Definition 4.3 *Side effects* for the current experiment are all the other experiments whose output values are complementary to the current experiment.

Definition 4.4 An *on-path side effect* is one that differs from the current experiment in only the vertex variables with assigned values.

Definition 4.5 An *off-path side effect* is one that differs from the current experiment in not only the vertex variables with assigned values but also some don't care variables.

Definition 4.6 Two off-path side effects are *disjoint* if their don't care terms can be set independently; otherwise they are *joint* side effects.

Definition 4.7 A 0-experiment is an experiment that has a 0 outcome. A 1-experiment is an experiment that has a 1 outcome.

Theorem 4.2 All the detectable faults of an experiment can be detected if the test set is formed with the unknowns assigned values that select side effects.

The objective in this approach is to exercise every experiment to verify that all paths through the circuit work correctly. In addition, don't care terms that correspond to unknown vertex values are set in such a way that all detectable wrong paths can be detected.

Example The BDD in Figure 4.27 has the following experiments:

0-experiments: $A,B,C = 00x, 011, 1x0$
1-experiments: $A,B,C = 010, 1x1$

When the current experiment is $A,B,C = 011$, the expected output is 0. An onpath side effect is $A,B,C = 010$. This means that an SA0 fault at input C will cause a 1 at the output, hence it will be detected. For the 0-experiment $A,B,C = 00x$, the expected result is 0. An off-path side effect is 010; it causes the 1-edge to be taken from B. An SA1 at input B causes the 1-edge to be taken, so if input C is set to 0, the circuit responds with a 1, and the SA1 is detected. ■■

Theorem 4.3 For a binary decision diagram that has m 0-experiments and n 1-experiments, the upper and lower bounds for the size of its test set N are $2mn$ and $m + n$, respectively.

Proof In the worst case, every 1-experiment is an off-path side effect for the 0-experinents, and all of them are needed to detect vertices with unknown values. Thus, the size of the test set for the 0-experiments is mn. Similarly, it is mn for the 1-experiments, so $N = 2mn$. If all the side effects for 0-experiments and 1-experiments are on-path

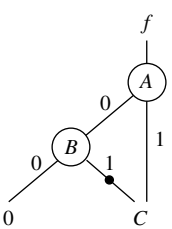


Figure 4.27 BDD for experiments.

side effects, then these $m + n$ experiments define all the tests, and that is the lower bound.

Because BDDs represent the behavior of a circuit, without regard to how it is constructed, structural information detailing the circuit's internal organization can easily be overlooked. Consider the BDD for the ripple-carry adder shown in Figure 4.26. This BDD could be used to characterize the behavior of a carry-lookahead adder. But the lack of detail describing the implementation of the circuit can lead to some stuck-at faults being overlooked. In their article, Chang et al. confirm that BDDs for the ripple-carry adder, when used to generate tests for the carry-lookahead adder, miss some of the faults that are detected when using the more detailed BDD.[24]

4.14 SUMMARY

The purpose of ATPG is to create test vectors that sensitize enough unique signal paths through a circuit, to observable outputs, such that if the circuit passes the test, there is a high degree of confidence that the circuit is free of defects. It is desirable to accomplish this with the smallest possible number of test vectors so that the circuit spends the least possible amount of time on the tester.

Numerous methods have been devised to create test patterns for combinational logic. The methods range from topological to algebraic and they date from the early 1960s to the present. Some are effective and widely used, whereas others are primarily of academic interest. They all have one thing in common: Their objective is to create input patterns that cause the output response of a circuit to depend on the presence or absence of some hypothesized set of faults. Secondary objectives, not explicitly addressed in this chapter, but which will be addressed in more detail in later chapters, include:

Thoroughness (comprehensiveness)

Ease of use

Ease of implementation

Fault resolution (ability to identify *which* fault occurred)

Efficiency (minimum number of vectors to achieve coverage goals)

Among the path tracing methods, the sensitized path was first to appear. R. D. Eldred advocated modeling stuck-at faults and creating specific tests to detect these faults. However, the first suggestion for the use of the sensitized path is attributed to an unidentified attendee at a conference at the University of Michigan in 1961. Path sensitizing programs had already been well developed by C. B. Steiglitz and others[1] when the D-algorithm was introduced in 1966. The D-algorthm provides a formal calculus for computing test vectors, and it explores the entire solution space, if necessary; hence it qualifies as an algorithm. In fact, it was the first method shown to be an algorithm. It relies on PDCFs and propagation D-cubes that are derived from a truth table and which can be created for any reasonable-sized entry in a cell library.

In combinational arrays that have many repetitive structures, it may be more economical to create custom-tailored primitives than to decompose library entries into their gate-level constituents.

PODEM enjoys an advantage over the D-algorithm on circuits that contain a great deal of reconvergent fanout, particularly circuits such as parity checkers that contain large numbers of XOR gates, because the basic D-algorithm will frequently attempt to justify specific logic values on inputs to XORs when either value is adequate. PODEM is elegant in its simplicity and quite straightforward to implement. However, that elegance comes at a price. FAN identifies situations where PODEM makes unnecessary calculations and adds enhancements to eliminate them. The goal of FAN is to reduce the number of backtracks and reduce the amount of processing time for each backtrack. Some of these techniques, such as the forward and back imply operations, are adopted directly from the D-algorithm. Socrates identifies additional enhancements, resulting in further performance gains. The critical path, employed in the LASAR test generation system, enjoyed commercial success in the era when PCBs were made up of SSI, MSI, and LSI (small-,medium-, and large-scale integration) parts.

It is interesting to contrast the different methods. LASAR works back from the outputs, whereas PODEM works forward from the inputs. The D-algorithm starts at the point of a fault, in the middle of a circuit, and propagates forward to an output, while working backwards to justify assignments as it proceeds. The D-algorithm can be implemented so as to perform complete justification back to the input pins for every step of the propagation, or, alternatively, it can be implemented so as to propagate completely to the outputs and save all justification steps until it has completed the propagation phase. Different circuits may favor one or another of these justification approaches.

Algebraic techniques are quite thorough and complete, it is possible to get a closed-form expression that describes the entire solution space for a given stuck-at fault. They demonstrate the disparate ways in which to approach and solve a problem. However, converting a netlist into Boolean equations (for both the fault-free and faulty circuits) and performing an exclusive-OR on these two representations is a nontrivial task. Boolean satisfiability lies somewhere between the pure structural algorithms and the algebraic methods. It translates the netlist to a conjunctive normal form. A search for a solution then involves finding a consistent set of assignments for the binary clauses while the ternary clauses serve as constraints.

BDDs have been growing in popularity in recent years, because of their widespread applicability to several areas of electronic design automation. It is interesting to note that one of the earliest applications of BDDs was to implement ATPG algorithms. The basic BDD functions, Reduce, Apply, Traverse, and so on, have applicability to simulation, as was seen in Chapter 2, and they have applicability to ATPG. Given a ROBDD for the fault-free and faulty circuits, the XOR operation is straightforward, and there are no backtracks. Furthermore, in contrast to other methods, the amount of CPU time does not depend on whether or not the fault is detectable.

It is important to note that, while the various ATPG algorithms each has advocates claiming that their method is superior to all others, in the final analysis, performance of a given algorithm often depends on how it was implemented. A method may, in theory, be an algorithm, but if the program takes shortcuts, it may no longer be an algorithm. Furthermore, ATPG is one of those applications where 95% of the CPU time is spent in 5% of the code. It is not unusual for implementations of the same algorithm to differ in performance by a factor of two or more simply because one algorithm was implemented more efficiently than the other in that critical 5%. Benchmark circuits also influence the outcome of performance comparisons. For every algorithm there is a circuit that favors it, and there is another circuit that will reduce its performance to a crawl.

PROBLEMS

4.1 A 32-bit ALU is to be tested with an exhaustive test (i.e., applying all possible input combinations). The ALU has 70 inputs: two 32-bit ports, a carry-in, and five op-codes to select the operation to be performed. If a tester can apply stimuli at the rate of one vector every 10ns, how long will it take to apply the entire test?

4.2 A four-input AND gate is exercised with the following test pattern set, which causes all of the inputs and the output to switch in both directions: (1,0,0,0), (0,1,0,0), (0,0,1,0), (0,0,0,1), (1,1,1,1). Assuming SA1 faults on each of the input pins, and SA0 and SA1 faults on the output, what is the fault coverage?

4.3 For the example in Section 4.3.1, the cube (1, X, 1, 0) is a prime cube but it is not an extremal. Why?

4.4 List the PDCFs for a four-input NOR gate. Assume faults on all inputs and two faults on the output.

4.5 Find a function for which 2^{2n-1} distinct propagation D-cubes exist.

4.6 How many vertices are represented by the vector (1, 0, D, X, 0, X, $\overline{\text{D}}$, X)?

4.7 Given the following cubes: a = (1, 0, X), b = (X, 0, 0), c = (1, 1, 1), d = (X, X, 1), e = (X, X, 0).

(a) Determine which cubes contain others.

(b) Perform all pairwise intersections, using the table in Section 4.3.1.

4.8 Two shipments of ICs have become mixed up. The ICs implement the functions F and F^*, defined below. How would you tell them apart if you had access to a tester?

$$F = a \cdot b \cdot c + b \cdot \overline{c} \cdot d + \overline{a} \cdot \overline{b} \cdot d + \overline{a} \cdot c \cdot \overline{d}$$

$$F^* = (a + b) \cdot (c + d)$$

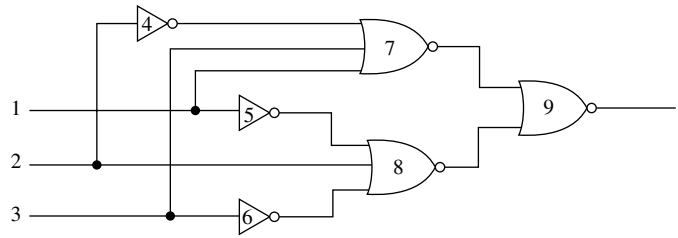


Figure 4.28 Creating the D-chains.

4.9 During creation of a sensitized path, two or more D and/or $\overline{D}$ signals converge on inputs to a primitive element. If the propagation table does not contain cubes with multiple D and $\overline{D}$ signals, explain how you would determine what value from the set $\{0,1,D,\overline{D}\}$ would propagate to the output.

4.10 Using the D-algorithm, create a test for a SA0 fault on the bottom input of gate 7 in the circuit of Figure 4.28. Show the D-chains for each step of the process.

4.11 Given an AND gate that drives five destination gates, what is the maximum number of propagation paths that D-algorithm must explore before it can conclude that a solution does not exist?

4.12 Create propagation D-cubes for the odd parity equation $Odd = I_1 \oplus I_2 \oplus I_3 \oplus I_4$, where $\oplus$ denotes exclusive-OR.

4.13 The following user defined primitive (UDP) describes a 2-to-1 multiplexer.

```
primitive MUX2_1 (Q, A, B, SEL);
output Q;
input A, B, SEL;
    table
    //  A   B   SEL :   Q
        0   0   ?   :   0 ;
        1   1   ?   :   1 ;
        0   ?   0   :   0 ;
        1   ?   0   :   1 ;
        ?   0   1   :   0 ;
        ?   1   1   :   1 ;
    endtable
endprimitive
```

Using the UDP, create the PDCFs and propagation D-cubes. The 2-to-1 multiplexer has reconvergent fanout inside the circuit, resulting in a fault that may not be detected by test vectors that detect faults on the pins. How would you compensate for that?

4.14 Create PDCFs and propagation D-cubes for the full-adder characterized by the following two verilog equations. First create truth tables for *Sum* and *Carry*. Then, from the truth tables, create the PDCFs and propagation D-cubes.

$$Sum = A \wedge B \wedge Cin;$$

$$Carry = A \,\&\, B \mid A \,\&\, Cin \mid B \,\&\, Cin;$$

4.15 In Section 4.5 it was stated that the subscripted D-algorithm could find many other tests for the indicated faults on gate 16 of Figure 4.10. Find as many solutions as you can.

4.16 Apply the pattern (11010) to the circuit in Figure 4.1 and use testdetect to find all stuck-at faults on gate inputs and outputs that are detected by that pattern.

4.17 Using PODEM, find a test for the indicated fault in Figure 4.29.

4.18 Use PODEM to find a test for a SA1 on the top input to gate *D* in Figure 4.1.

4.19 The bottom input to gate *G* in Figure 4.1 is redundant. Using PODEM, prove that the input is redundant.

4.20 Given a two-input XOR gate, explain what happens when sensitized values arrive at both inputs. Consider all four cases: (D,D), (D,$\overline{\text{D}}$), ($\overline{\text{D}}$,D), ($\overline{\text{D}}$,$\overline{\text{D}}$).

4.21 Create a NAND-equivalent version of the circuit in Figure 4.1, use critical path to generate tests for all four input stuck-at faults on the NOR labeled *J*. Note that the bubble on its third input implies that the input must be tested for a fault of the opposite polarity from the others.

4.22 Use FAN to generate a test for a SA0 on the output of gate *B* in the circuit of Figure 4.3.

4.23 Finish the computations for the Boolean difference example at the end of Section 4.11.

4.24 Use the Boolean difference to find a test for a fault on the middle input to gate 8 in Figure 4.20.

4.25 In the example used to describe Boolean satisfiability, the initial formula reduced to $(\overline{A} + \overline{B})$ after all implications were performed. Show the details; that is, prove that this result is correct.

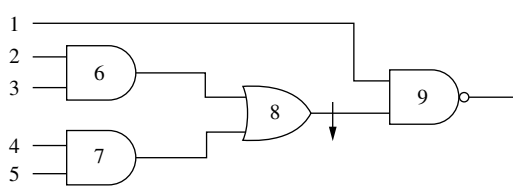


Figure 4.29 Finding a test with PODEM.

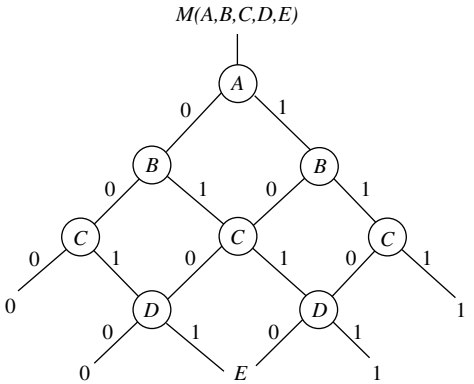


Figure 4.30 0- and 1-experiments.

4.26 Use Boolean satisfiability to find a test for a SA0 on the bottom input to gate 7 in Figure 4.22.

4.27 Two equations were given for the circuit in Figure 4.22, one for the good circuit, g, and another for the faulted circuit, f. Use the Apply algorithm to create ROBDDs B_g and B_f. Then compute Apply($\oplus$, B_g, B_f).

4.28 In the 3-of-5 majority function *M(A, B, C, D, E)* illustrated in Figure 4.30:

list all of the 0-experiments and all of the 1-experiments,

determine the bounds on the number of tests required,

from the BDD, generate the tests required to fully test the circuit.

4.29 Given the equation $F = D \cdot ((A \cdot B) + (\overline{A} \cdot C))$, create a BDD with A as the root and repeat the previous problem.

4.30 The following equations describe a carry look-ahead (CLA):

$$C_{n+x} = G_0 + P_0 C_n$$
$$C_{n+y} = G_1 + P_1 G_0 + P_1 P_0 C_n$$
$$C_{n+z} = G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 C_n$$
$$G = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0$$
$$P = P_3 P_2 P_1 P_0$$

Create a BDD for the CLA. Show how to connect it with four of the BDDs in Figure 2.35(g) to form a 16-bit adder.

4.31 Using the circuit in Figure 4.1, generate the ROBDD corresponding to a SA0 on input 2 of gate M. Then use Apply to compute the XOR of that ROBDD and the ROBDD in Figure 2.44. Reduce the resulting OBDD and convert it to a closed form Boolean expression.

REFERENCES

1. Case, P. W. et al., Design Automation in IBM, *IBM J. Res. Dev.*, Vol. 25, No. 5, September 1981, pp. 631–646.
2. Schneider, P. R., On the Necessity to Examine D-chains in Diagnostic Test Generation—An Example, *IBM J. Res. Dev.*, Vol. 10, No. 1, January 1967, p. 114.
3. Roth, J. P., Diagnosis of Automata Failures: A Calculus and a Method, *IBM J. Res. Dev.*, Vol. 10. No. 4, July 1966, pp. 278–291.
4. Roth, J. P. et al., Programmed Algorithms to Compute Tests to Detect and Distinguish Between failures in Logic Circuits, *IEEE Trans. Comput.*, Vol. EC-16, No. 5, October 1967, pp. 567–580.
5. Roth, J. P., *Computer Logic, Testing, and Verification*, Chapter 3, Computer Science Press, Potomac, MD, 1980.
6. Benmehrez, C., and J. F. McDonald, Measured Performance of a Programmed Implementation of the Subscripted D-algorithm, *Proc. 20th Des. Autom. Conf.*, 1983, pp. 308–315.
7. Kirkland, Tom, and M. R. Mercer, Algorithms for Automatic Test Pattern Generation, *IEEE Des. Test*, Vol. 5, No. 3, June 1988, pp. 43–55.
8. McDonald, J. F., and C. Benmehrez, Test Set Reduction Using the Subscripted D-algorithm, *Proc. 1983 Int. Test Conf.*, October 1983, pp. 115–121.
9. Goel, P., An Implicit Enumeration Algorithm to Generate Tests for Combinational Logic Circuits, *IEEE Trans. Comput.*, Vol. C-30, No. 3, March 1981, pp. 215–222.
10. Lawler, E. W., and D. E. Wood, Branch-and-Bound Methods—A Survey, *Oper. Res.*, Vol. 14, 1966, pp. 669–719.
11. Fujiwara, H., and T. Shimono, On the Acceleration of Test Generation Algorithms, *IEEE Trans. Comput*., Vol. C-32, No. 12, December 1983, pp. 1137–1144.
12. Schulz, M. H. et al., SOCRATES: A Highly Efficient Automatic Test Pattern Generation System, *IEEE Trans. CAD*, Vol. 7, No. 1, January 1988, pp. 126–137.
13. Wang, David T., An Algorithm for the Generation of Test Sets for Combinational Logic Networks, *IEEE Trans. Comp.*, Vol. C-24, No. 7, July 1975, pp. 742–746.
14. Thomas, J. J., Automated Diagnostic Test Programs for Digital Networks, *Computer Des.*, August 1971, pp. 63–67.
15. Abramovici, M. et al., Critical Path Tracing—An Alternative to Fault Simulation, *Proc. 20th Des. Automat., Conf.*, 1983, pp. 214–220.
16. Abramovici, M. et al., Critical Path Tracing—An Alternative to Fault Simulation, *IEEE Des. Test Mag.*, Vol. 1, No. 1, February 1984, pp. 83–93.
17. Hong, S. J., Fault Simulation Strategy for Combinational Logic Networks, *Proc. 8th Int. Symp. on Fault-Tolerant Computing*, 1978, pp. 96–99.
18. Miczo, A., Concurrent Fault Simulation: Some Performance Measurements, unpublished paper.
19. Sellers, F. F. et al., Analyzing Errors with the Boolean Difference, *IEEE Trans. Comput.*, Vol. C-17, No. 7, July 1968, pp. 676–683.
20. Akers, S. B., On a Theory of Boolean Functions, *J. SIAM*, Vol. 7, December 1959.
21. Larrabee, T., Test Pattern Generation Using Boolean Satisfiability, *IEEE Trans. CAD.*, January 1992, pp. 4–15.

22. Abadir, M. S., and H. K. Reghbati, Test Generation for LSI: A Case Study, *Proc. 21st Des. Autom. Conf.*, 1984, pp. 180–195.
23. *The Am2900 Family Data Book,* Advanced Micro Devices, Inc., Sunngvale, CA, 1979.
24. Chang, H. P. et al., Structured Functional Level Test Generation Using Binary Decision Diagrams, *Proc. 1986 Int. Test Conf.*, pp. 97–104.

CHAPTER 5

Sequential Logic Test

5.1 INTRODUCTION

The previous chapter examined methods for creating sensitized paths in combinational logic extending from stuck-at faults on logic gates to observable outputs. We now attempt to create tests for sequential circuits where the outputs are a function not just of present inputs but of past inputs as well. The objective will be the same: to create a sensitized path from the point where a fault occurs to an observable output. However, there are new factors that must be taken into consideration. A sensitized path must now be propagated not only through logic operators, but also through an entirely new dimension—time. The time dimension may be discrete, as in synchronous logic, or it may be continuous, as in asynchronous logic.

The time dimension was ignored when creating tests for faults in combinational logic. It was implicitly assumed that the output response would stabilize before being measured with test equipment, and it was generally assumed that each test pattern was independent of its predecessors. As will be seen, the effects of time cannot be ignored, because this added dimension greatly influences the results of test pattern generation and can complicate, by orders of magnitude, the problem of creating tests. Assumptions about circuit behavior must be carefully analyzed to determine the circumstances under which they prevail.

5.2 TEST PROBLEMS CAUSED BY SEQUENTIAL LOGIC

Two factors complicate the task of creating tests for sequential logic: memory and circuit delay. In sequential circuits the signals must not only be logically correct, but must also occur in the correct time sequence relative to other signals. The test problem is further complicated by the fact that aberrant behavior can occur in sequential circuits when individual discrete components are all fault-free and conform to their manufacturer's specifications. We first consider problems caused by the presence of memory, and then we examine the effects of circuit delay on the test generation problem.

Digital Logic Testing and Simulation, *Second Edition*, by Alexander Miczo
ISBN 0-471-43995-9 Copyright © 2003 John Wiley & Sons, Inc.

5.2.1 The Effects of Memory

In the first chapter it was pointed out that, for combinational circuits, it was possible (but not necessarily reasonable) to create a complete test for logic faults by applying all possible binary combinations to the inputs of a circuit. That, as we shall see, is not true for circuits with memory. They may not only require more than 2^n tests, but are also sensitive to the *order* in which stimuli are applied.

Test Vector Ordering The effects of memory can be seen from analysis of the cross-coupled NAND latch [cf. Figure 2.3(b)]. Four faults will be considered, these being the input SA1 faults on each of the two NAND gates (numbering is from top to bottom in the diagram). All four possible binary combinations are applied to the inputs in ascending order—that is, in the sequence ($\overline{\text{Set}}$, $\overline{\text{Reset}}$) = {(0,0), (0,1), (1,0), (1,1)}. We get the following response for the fault-free circuit (FF) and the circuit corresponding to each of the four input SA1 faults.

Input		Output				
$\overline{\text{Set}}$	$\overline{\text{Reset}}$	FF	1	2	3	4
0	0	1	0	1	1	1
0	1	1	0	1	1	1
1	0	0	0	0	0	1
1	1	0	0	0	1	1

In this table, fault number 2 responds to the sequence of input vectors with an output response that exactly matches the fault-free circuit response. Clearly, this sequence of inputs will not distinguish between the fault-free circuit and a circuit with input 2 SA1.

The sequence is now applied in the exact opposite order. We get:

Input		Output				
$\overline{\text{Set}}$	$\overline{\text{Reset}}$	FF	1	2	3	4
1	1	?	?	0	1	?
1	0	0	0	0	0	?
0	1	1	0	1	1	1
0	0	1	0	1	1	1

The Indeterminate Value When the four input combinations are applied in reverse order, question marks appear in some table positions. What is their significance? To answer this question, we take note of a situation that did not exist when dealing only with combinational logic; the cross-coupled NAND latch has *memory.* By virtue of feedback present in the circuit, it is able to remember the value of a signal that was applied to the set input even after that signal is removed.

Because of the feedback, neither the $\overline{\text{Set}}$ nor the $\overline{\text{Reset}}$ line need be held low any longer than necessary to effectively latch the circuit. However, when power is first applied to the circuit, it is not known what value is contained in the latch. How can circuit behavior be simulated when it is not known what value is contained in its memory?

In real circuits, memory elements such as latches and flip-flops have indeterminate values when power is first applied. The contents of these elements remain indeterminate until the latch or flip-flop is either set or reset to a known value. In a simulation model this condition is imitated by initializing circuit elements to the indeterminate X state. Then, as seen in Chapter 2, some signal values can drive a logic element to a known state despite the presence of indeterminate values on other inputs. For example, the AND gate in Figure 2.1(c) responds with a 0 when any single input receives a 0, regardless of what values are present on other inputs. However, if a 1 is applied while all other inputs are at X, the output remains at X.

Returning to the latch, the first sequence began by applying 0s to both inputs, while the second sequence began by applying 1s to both inputs. In both cases the internal nets were initially indeterminate. The 0s in the first sequence were able to drive the latch to a known state, making it possible to immediately distinguish between correct and incorrect response. When applying the patterns in reverse order, it took longer to drive the latch into a state where good circuit response could be distinguished from faulty circuit response. As a result, only one of the four faults is detected, namely, fault 1. Circuits with faults 2 and 3 agree with the good circuit response in all instances where the good circuit has a known response. On the first pattern the good circuit response is indeterminate and the circuit with fault 2 responds with a 0. The circuit with fault 3 responds with a 1. Since it is not known what value to expect from the good circuit, there is no way to decide whether the faulted circuits are responding correctly.

Faulted circuit 4 presents an additional complication. Its response is indeterminate for both the first and second patterns. However, because the good circuit has a known response to pattern 2, we do know what to look for in the good circuit, namely, the value 0. Therefore, if a NAND latch is being tested with the second set of stimuli, and it is faulted with input 4 SA1, it might come up initially with a 0 on its output when power is applied to the circuit, in which case the fault is not detected, or it could come up with a 1, in which case the fault will be detected.

Oscillations Another complication resulting from the presence of memory is oscillations. Suppose that we first apply the test vector (0,0) to the cross-coupled NAND latch. Both NAND gates respond with a logic 1 on their outputs. We then apply the combination (1,1) to the inputs. Now there are 1s on both inputs to each of the two NAND gates—but not for long. The NAND gates transform these 1s into 0s on the outputs. The 0s then show up on the NAND inputs and cause the NAND outputs to go to 1s. The cycle is repetitive; the latch is oscillating. We do not know what value to expect on the NAND gate outputs; the latch may continue to oscillate until a different stimulus is applied to the inputs or the oscillations may eventually subside.

If the oscillations do subside, there is no practical way to predict, from a logic description of the circuit, the final state into which the latch settles. Therefore, the NAND outputs are set to the indeterminate X.

Probable Detected Faults When we analyzed the effectiveness of binary sequences applied to the NAND latch in descending order, we could not claim with certainty that stuck-at fault number 4 would be detected. Fortunately, that fault is detected when the vectors are applied in ascending order. In other circuits the ambiguity remains. In Figure 2.4(b) the Data input is complemented and both true and complement values are applied to the latch. Barring the presence of a fault, the latch will not oscillate. However, when attempting to create a test for the circuit, we encounter another problem. If the $\overline{\text{Enable}}$ signal is SA1, the output of the inverter driven by $\overline{\text{Enable}}$ is permanently at 0 and the NAND gates driven by the inverter are permanently in a 1 state; hence the faulted latch cannot be initialized to a known state. Indeterminate states were set on the latch nodes prior to the start of test pattern generation and the states remain indeterminate for the faulted circuit. If power is applied to the fault-free and faulted latches, the circuits may just happen to come up in the same state.

The problem just described is inherent in any finite-state machine (FSM). The FSM is characterized by a set of states $Q = \{q_1, q_2, ..., q_s\}$, a set of input stimuli $I = \{i_1, i_2, ..., i_n\}$, another set $Y = \{y_1, y_2, ..., y_m\}$ of output responses, and a pair of mappings

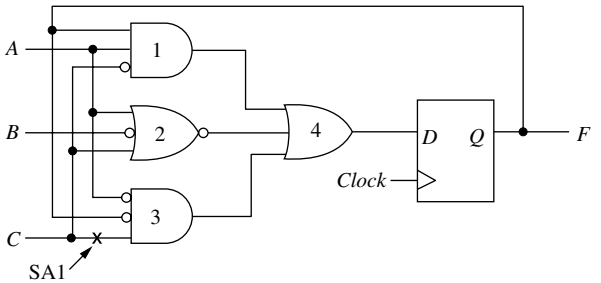
$$M : Q \times I \rightarrow Q$$

$$Z : Q \times I \rightarrow Y$$

These mappings define the next state transition and the output behavior in response to any particular input stimulus. These mappings assume knowledge of the current state of the FSM at the time the stimulus is applied. When the initial stimulus is applied, that state is unknown unless some independent means such as a reset exists for driving the FSM into a known state.

In general, if there is no independent means for initializing an FSM, and if the Clock or Enable input is faulty, then it is not possible to apply just a single stimulus to the FSM and detect the presence of that fault. One approach used in industry is to mark a fault as a *probable detect* if the fault-free circuit drives an output pin to a known logic state and the fault causes that same pin to assume an unknown state.

The industry is not in complete agreement concerning the classification of probable detected faults. While some test engineers maintain that such a fault is likely to eventually become detected, others argue that it should remain classified as undetected, and still others prefer to view it as a probable detect. If the probable detected fault is marked as detected, then there is a concern that an ATPG may be designed to ignore the fault and not try to create a test for it in those situations where a test exists.



A	B	F
0	0	C
0	1	$\overline{Q}$
1	0	Q
1	1	0

Figure 5.1 Initialization problem.

The Initialization Problem Consider the circuit of Figure 5.1. During simulation, circuit operation begins with the D flip-flop in an unknown state. In normal operation, when the input combination $A = B = C = 0$ is applied and the flip-flop is clocked, the Q output switches to 0. The flip-flop can then be clocked a second time to obtain a test for the lower input of gate 3 SA1. If it is SA1, the expected value is $Q = 1$; and if it is fault-free, the expected value is $Q = 0$.

Unfortunately, the test has a serious flaw! If the lower input to gate 3 is SA1, the output of the flip-flop at the end of the first clock period is indeterminate because the value at the middle input to gate 3 is initially indeterminate. It is driven by the flip-flop that has an indeterminate value. After a second clock pulse the value at Q will remain at X; hence it may agree with the good circuit response despite the presence of the fault. The fallacy lies in assuming correct circuit behavior when setting up the flip-flop for the test. We depended upon correct behavior of the very net that we are attempting to test when setting up a test to detect a fault on that net.

To correctly establish a test, it is necessary to assume an indeterminate value from the flip-flop. Then, from the D-algorithm, we know that the flip-flop must be driven into the 0 state, without depending on the input to gate 3 that is driven by the flip-flop. The flip-flop value can then be used in conjunction with the inputs to test for the SA1 on the lower input of gate 3. In this instance, we can set $A = C = 0$, $B = 1$. Then a 1 can be clocked into the flip-flop from gate 2. This produces a 0 on the output of the flip-flop which can then be used with the assignment $A = B = 0$ to clock a 0 into the flip-flop. Now, with $Q = 0$ and $A = B = C = 0$, another clock causes $\overline{\text{D}}$ to appear on the output of the flip-flop.

Notice that input C was used, but it was used to set up gate 2. If input C were faulted in such a way as to affect both gates 2 and 3, then it could not have been used to set up the test.

5.2.2 Timing Considerations

Until now we have assumed that erroneous behavior on circuit outputs was the result of *logic* faults. Those faults generally result from actual physical defects such as opens or shorts, or incorrect fabrication such as an incorrect connection or a wrong

component. Unfortunately, this assumption, while convenient, is an oversimplification. An error may indeed be a result of one or more logic faults, but it may also be the case that an error occurs and none of the above situations exists.

Defects exist that can prevent an element from behaving in accordance with its specifications. Faults that affect the performance of a circuit are referred to as *parametric* faults, in contrast to the logic faults that have been considered up to this point. Parametric faults can affect voltage and current levels, and they can affect gain and switching speed of a circuit. Parametric faults in components can result from improper fabrication or from degradation as a consequence of a normal aging process. Environmental conditions such as temperature extremes, humidity, or mechanical vibration can accelerate the degradation process.

Design oversights can produce symptoms similar to parametric faults. Design problems include failure to take into account wire lengths, loading of devices, inadequate decoupling, and failure to consider worst-case conditions such as maximum or minimum voltages or temperatures over which a device may be required to operate. It is possible that none of these factors may cause an error in a particular design in a well-controlled environment, and yet any of these factors can destabilize a circuit that is operating under adverse conditions. Relative timing between signal paths or the ability of the circuit to drive other circuits could be affected.

Intermittent errors are particularly insidious because of their rather elusive nature, appearing only under particular combinations of circumstances. For example, a logic board may be designed for nominal signal delay for each component as a safety margin. Statistically, the delays should seldom accumulate so as to exceed a critical threshold. However, as with any statistical expectation, there will occasionally be a circuit that does exceed the maximum permissible value. Worse still, it may work well at nominal voltages and /or temperatures and fail only when voltages and/or temperatures stray from their nominal value. A new board substituted for the original board may be closer to tolerance and work well under the degraded voltage and/or temperature conditions. The original board may then, when checked at a depot or a board tester under ideal operating conditions, test satisfactorily.

Consider the effects of timing variations on the delay flip-flop of Figure 2.7. Correct operation of the flip-flop requires that the designer observe minimal setup and hold times. If propagation delay along a signal path to the Data input of the flip-flop is greater than estimated by the designer, or if parametric faults exist, then the setup time requirement relative to the clock may not be satisfied, so the clock attempts to latch the signal while it is still changing. Problems can also occur if a signal arrives too soon. The hold time requirement will be violated if a new signal value arrives at the data input before the intended value is latched up in the flip-flop. This can happen if one register directly feeds another without any intervening logic.

That logic or parametric faults can cause erroneous operation in a circuit is easy to understand, but digital test problems are further compounded by the fact that errors can occur during operation of a device when its components behave as intended. Elements used in the fabrication of digital logic circuits contain delay. Ironically, although technologists constantly try to create faster circuits and reduce delay, sequential logic circuits cannot function without delay; circuits depend both

on correct logic operation of circuit components and on correct relative timing of signals passing through the circuit. This delay must be taken into account when designing and testing circuits.

Suppose the inverter driven by the Data input in the gated latch circuit of Figure 2.4(b) has a delay of *n* nanoseconds. If the Data input makes a 0-to-1 transition followed by a 0-to-1 transition on the Enable approximately *n* nanoseconds later, the two cross-coupled NAND gates see an input of (0,0) for about *n* nanoseconds followed by an input of (1,1). This produces unpredictable results, as we have seen before. The problem is caused by the delay in the inverter. A solution to this problem is to put a buffer in the noninverting signal path so the Data and $\overline{\text{Data}}$ signals reach the NANDs at about the same time.

In each of the two circuits just cited, the delay flip-flop and the latch, a race exists. A *race* is a condition wherein two or more signals are changing simultaneously in a circuit. The race may be caused by multiple simultaneous input signal changes, or it may be the result of a single signal change that follows two or more paths from a fanout point. Note that any time we have a latch or flip-flop we have a race condition, since these devices will always have at least one element whose signal both goes outside the device and feeds back to an input of the latch or flip-flop. Races may or may not affect the behavior of a circuit. A *critical race* exists if the behavior of a circuit depends on the outcome of the race. Such races can produce unanticipated and unwanted results.

Hazards can also cause sequential circuits to behave in ways that were not intended. In Section 2.6.4 the consequences of several kinds of hazards were considered. Like timing problems, hazards can be extremely difficult to diagnose because their effect on a circuit may depend on other factors, such as marginal voltages or an operating temperature that is within specification but borderline. Under optimal conditions, a glitch caused by a hazard may not contain enough energy to cause a latch to switch state; but under the influence of marginal operating conditions, this glitch may have sufficient energy to cause a latch of flip-flop to switch states.

5.3 SEQUENTIAL TEST METHODS

We now examine some methods that have been developed to create tests for sequential logic. The methods described here, though not a complete survey, are representative of the methods described in the literature and range from quite simple to very elaborate. To simplify the task, we will confine our attention in this chapter to errors caused by logic faults. Intermittent errors, such as those caused by parametric faults or races and hazards, will be discussed in subsequent chapters.

5.3.1 Seshu's Heuristics

Some of the earliest documented attempts at automatically generating test programs for digital circuits were published in 1965 by Sundaram Seshu.[1] These

made use of a collection of heuristics to generate trial patterns or sequences of patterns that were then simulated in order to evaluate their effectiveness. Seshu identified four heuristics for creating test patterns. The test patterns created were actually trial test patterns whose effectiveness was evaluated with the simulator. If the simulator indicated that a given pattern was ineffective, the pattern was rejected and another trial pattern was selected and evaluated. The four heuristics employed were

Best next or return to good
Wander
Combinational
Reset

We briefly describe each of these:

Best Next or Return to Good The best next or return to good begins by selecting an initial test pattern, perhaps one that resets the circuit. Then, given a $(j-1)$st pattern, the jth pattern is determined by simulating all next patterns, where a *next pattern* is defined as any pattern that differs from the present pattern in exactly one bit position. The next pattern that gives best results is retained. Other patterns that give good results are saved in a pushdown stack. If no trial pattern gives satisfactory results at the jth step, then the heuristic selects some other $(j-1)$st pattern from the stack and tries to generate the jth vector from it. If all vectors in the stack are discarded, the heuristic is terminated. A pattern may give good results when initially placed on the stack but no longer be effective when simulating a sequential circuit because of the feedback lines. When the pattern is taken from the stack, the circuit may be in an entirely different state from that which existed when the pattern was placed on the stack. Therefore, it is necessary to reevaluate the pattern to determine whether it is still effective.

Wander The wander heuristic is similar to the best next in that the $(j-1)$st vector is used to generate the jth by generating all possible next vectors. However, rather than maintain a stack of good patterns, if none of the trial vectors is acceptable, the heuristic "wanders" randomly. If there is no obvious choice for next pattern, it selects a next pattern at random. After each step in the wander mode, all next patterns are simulated. If there is no best next pattern, again wander at random and try all next patterns. After some fixed number of wander steps, if no satisfactory next pattern is found, the heuristic is terminated.

Combinational The combinational heuristic ignores feedback lines and attempts to generate tests as though the circuit were strictly combinational logic by using the path sensitization technique (Seshu's heuristics predate the D-algorithm). The pattern thus developed is then evaluated against the real circuit to determine if it is effective.

Reset The reset heuristic required maintaining a list of reset lines. This strategy toggles some subset of the reset lines and follows each such toggle by a fixed number of next steps, using one of the preceding methods, to see if any useful information is obtained.

The heuristics were applied to some rather small circuits, the circuit limits being 300 gates and no more than 48 each of inputs, outputs, and feedback loops. Additionally, the program could handle no more than 1000 faults. The best next or return to good was reported to be the most effective. The combinational was effective primarily on circuits with very few feedback loops. The system had provisions for human interaction. The test engineer could manually enter test patterns that were then fault simulated and appended to the automatically generated patterns. The heuristics were all implemented under control of a single control program that could invoke any of them and could later call back any of the heuristics that had previously been terminated.

5.3.2 The Iterative Test Generator

The heuristics of Seshu are easy to implement but not effective for highly sequential circuits. We next examine the iterative test generator (ITG)[2,3] which can be viewed as an extension to Seshu's combinational heuristic. Whereas Seshu treats a mildly sequential circuit as combinational by ignoring feedback lines, the iterative test generator transforms a sequential circuit into an *iterative array* by means of loop-cutting. This involves identifying and cutting feedback lines in the computer model of the circuit. At the point where these cuts are made, pseudo-inputs *SI* and pseudo-outputs *SO* are introduced so that the circuit appears combinational in nature. The new circuit *C* contains the pseudo-inputs and pseudo-outputs as well as the original primary inputs and primary outputs. This circuit, in Figure 5.2, is replicated p times and the pseudo-outputs of the ith copy are identified with the pseudo-inputs of the $(i + 1)$st copy.

The ATPG is applied to circuit *C* consisting of the p copies. A fault is selected in the jth copy and the ATPG tries to generate a test for the fault. If the ATPG assigns a logic value to a pseudo-input during justification, that assignment must be justified in the $(j - 1)$st copy. However, the ATPG is restricted from assigning values to the pseudo-inputs of the first copy. These pseudo-inputs must be assigned the X state. The

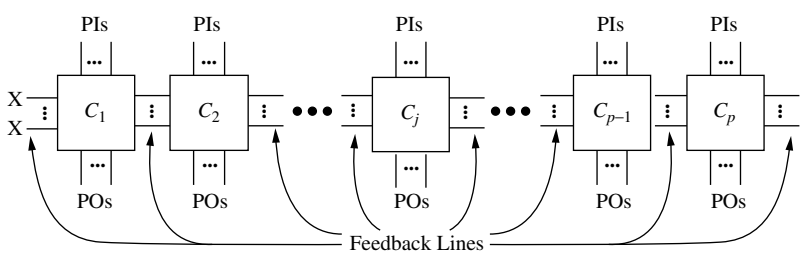


Figure 5.2 Iterative Array.

objective is to create a *self-initializing sequence*—that is, one in which all requirements on feedback lines are satisfied without assuming the existence of known values on any feedback lines at the start of the test sequence for a given fault. From the *j*th copy, the ATPG tries to propagate a D or $\overline{D}$ forward until, in some copy C_m, $m \leq p$, the D or $\overline{D}$ reaches a primary output or the last copy C_p is reached, in which case the test pattern generator gives up.

The first step in the processing of a circuit is to "cut" the feedback lines in the circuit model. To assist in this process, weights are assigned to all nets, subject to the rule that a net cannot be assigned a weight until all its predecessors have been assigned weights, where a *predecessor* to net *n* is a net connected to an input of the logic element that drives net *n*. The weights are assigned according to the following procedure:

1. Define for each net an intrinsic weight *IW* equal to its fanout minus 1.
2. Assign to each primary input a weight $W = IW$.
3. If weights have been assigned to all predecessors of a net, then assign a weight to that net equal to the sum of the weights of its predecessors plus its intrinsic weight.
4. Continue until all nets that can be weighted have been weighted.

If all nets are weighted, the procedure is done. If there are nets not yet weighted, then loops exist. The weighting process cannot be completed until the loops are cut, but in order to cut the loops they must first be identified and then points in the loops at which to make the cuts must be identified.

For a set of nets *S*, a subset S_1 of nets of *S* is said to be a *strongly connected component* (SCC), of *S* if:

1. For each pair of nets *l*, *m* in S_1 there is a directed path connecting *l* to *m*.
2. S_1 is a maximal set.

To find an SCC, select an unweighted net *n* and create from it two sets $B(n)$ and $F(n)$. The set $B(n)$ is formed as follows:

(a) Set $B(n)$ initially equal to $\{n\} \cup \{\text{all unweighted predecessors of } n\}$.
(b) Select $m \in B(n)$ for some *m* not yet processed.
(c) Add to $B(n)$ the unweighted predecessors of *m* not already contained in $B(n)$.
(d) If $B(n)$ contains any unprocessed elements, return to step b.

Set $F(n)$ is formed similarly, except that it is initially the union of *n* and its unweighted successors, where the *successors* of net *m* are nets connected to the outputs of gates driven by *m*. When selecting an element *m* from $F(n)$ for processing, its unweighted and previously unprocessed successors are added to $F(n)$. The intersection of $B(n)$ and F(n) defines an SCC.

Continue forming SCCs until all unweighted nets are contained in an SCC. At least one SCC must exist for which all predecessors—that is, inputs that originate from outside the loop—are weighted (why?). Once we have identified such an SCC, we make a cut and assign weights to all nets that can be assigned weights, then make another cut if necessary and assign weights, until all nets in S_1 have been weighted. The successor following the cut is assigned a weight that is one greater than the maximum weight so far assigned. Any other gates that can be assigned weights are assigned according to step 3 above. When the SCC has been completely processed, select another SCC (if any remain), using the same criteria, continuing until all SCCs have been processed.

The selection of a point in an SCC A at which to make a cut requires assignment of a period to each gate in A. The *period* for a gate k is the length of the shortest cycle containing k. Let B represent a subset of blocks of minimum period within A. If B is identical to A, then select a gate g in A that feeds a gate outside A and make a cut on the net connecting g with the rest of A.

If B is a proper subset of A, then consider the set U of nets in $A - B$ that have some predecessors weighted. Let $U_1 \subseteq$ U be the set of nearest successors of B in U. Then U_1 is the set of candidate nets, one of whose predecessors will be cut. Select an element in U_1 driven by a weighted net of minimal weight. Since the weights assigned to nets indicate relative ease or difficulty of controlling the nets, gates with input nets that have low weights will be easiest to control; hence a cut on a net feeding such a gate should cause the least difficulty in controlling the circuit.

Example The JK flip-flop of Figure 5.3 will be used to illustrate the cut process. First, according to step 1, an intrinsic weight is assigned to each net. (Each net number is identified with the number of the gate or primary input that drives it.)

1	2	3	4	5	6	7	8	9	10	11	12	13	14
2	0	2	0	2	1	0	0	1	1	0	0	1	1

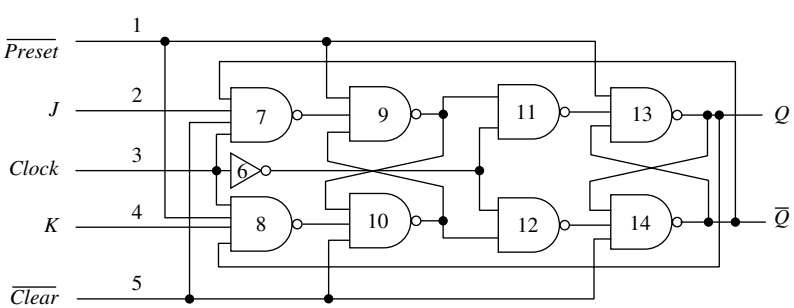


Figure 5.3 Cutting Loops.

Next, assign weights:

1	2	3	4	5	6	7	8	9	10	11	12	13	14
2	0	2	0	2	3								

From step 2 it is determined that line 6 must be assigned a weight of 3. At this point no other line can be assigned. The unweighted successors of the weighted lines consists of the set

$$A = \{7,8,9,10,11,12,13,14\}$$

A net is chosen and its SCC is determined. If net 7 is arbitrarily chosen, we find that its SCC is the entire set *A*. Since the SCC is the only loop in the circuit, all predecessors of the SCC are weighted so processing of the SCC can proceed.

We compute the periods of the nets in the SCC and find that nets 9, 10, 13, and 14 have period 2. Therefore, B = {9, 10, 13, 14}. In the set A − B = {7, 8, 11, 12} all nets have at least one weighted predecessor, so U = A − B. It also turns out that $U_1 = U$ in this case. A net in U_1 is selected that has a predecessor of minimal weight, say gate 7. A cut is made on net 14 between gate 14 and gate 7. The maximum weight assigned up to this point was 3. Therefore, we assign a weight of 4 to net 7. At this point weights cannot be assigned to any additional nets because loops still exist. The SCC is

$$A = \{8,9,10,11,12,13,14\}$$

The process is repeated, this time a cut is made from gate 13 to gate 8. A weight of 5 is assigned to net 8. This leaves two SCCs, $C = \{9,10\}$ and $D = \{13,14\}$. *C* must be chosen because *D* has unweighted predecessors. A cut is made from 9 to 10. A weight of 6 is assigned to net 10 and a weight of 2 + 4 + 6 + 1 = 13 to net 9. Weights can now be assigned to nets 11 and 12. Net 11 is assigned a weight of 13 + 3 + 0 = 16 and net 12 is assigned a weight of 9. Finally, a cut is made from 13 to 14. Net 14 is given the weight 17 and 13 is given the weight 36. ■■

The ITG will now be illustrated, using the circuit in Figure 5.4. The original circuit had one feedback line from the output of *J* to the input of *H* that was cut and replaced by a pseudo-input *SI* and a pseudo-output *SO*. The logic gates and primary inputs will be labeled with letters, and a subscript will be appended to the letters to indicate which copy of the replicated circuit is being referred to during the discussion.

We assume a SA1 fault on the output of gate *E*. A test for that fault requires a $\overline{D}$ on the net; so, starting with replica 2, we assign $A_2 = 1$. The output of *E* drives gates *F* and *G*, and here the ITG reverts to the sensitized path method, it chooses a single propagation path based on weights assigned during the cut process. The weights influence the path selection process: The objective is to try to propagate through the easiest apparent path. In this instance, the path through gate F_2 is selected. It requires a 0 from D_2, which in turn requires a 1 on input B_2. Propagation through K_2 requires a 1 from J_2 and hence 0s on input C_2 and gate H_2. The 0 on H_2 requires that

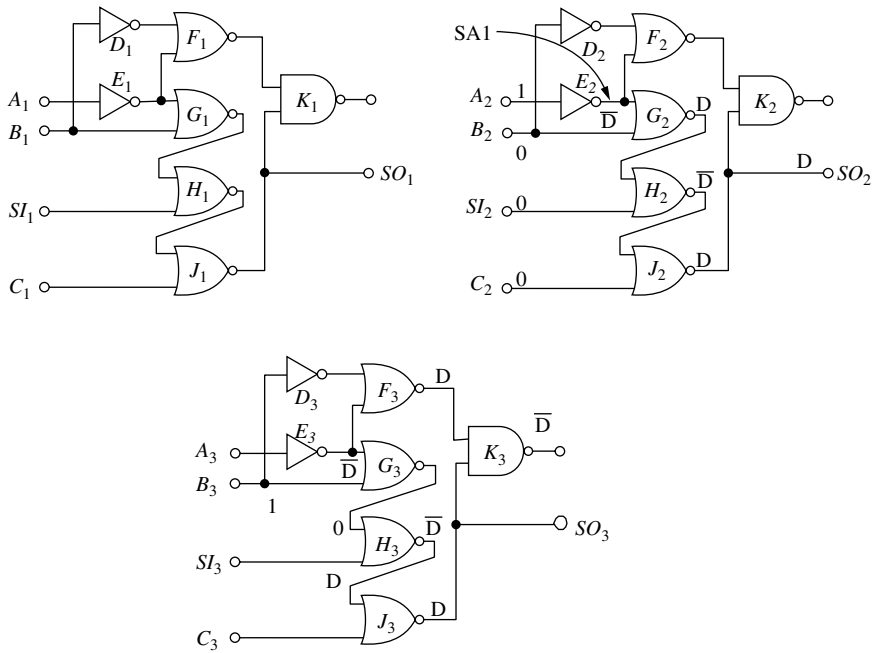


Figure 5.4 Iterated pseudo-combinational circuit.

pseudo-input SI_2 be a 1. The presence of a non-X value on a pseudo-input must be justified, so it is necessary to back up to the previous time image.

A 1 on the pseudo-output of J_1 implies 0s on both of its inputs. A 0 from H_1 requires a 1 on one of its inputs. We avoid SI_1 and try to assign $G_1 = 1$. That requires $E_1 = 0$, but E_1 is SA1. We cannot now, in this copy, assume that the output of E_1 is fault-free. Since it is assumed SA1, we could assign a $\overline{D}$, but that places a D and an X on H_1, a combination for which there is no entry in the D-algorithm intersection tables.

The other alternative is to assign a 1 to the pseudo-input, but that is no improvement because the same situation is encountered in the next previous time image. In practice, a programmed implementation may actually try to justify through the pseudo-input and go into a potential infinite loop. An implementation must therefore impose an upper limit on the number of previous time images. If all assignments are not justified by the time it reaches the limit, it must either give up on that fault or determine whether an alternative path exists through which to propagate the fault. In the present case, we can try to propagate through G_2.

Propagation through G_2 requires $B_2 = 0$. Then, propagation through H_2 requires a 0 on the pseudo input and propagation through J_2 requires $C_2 = 0$. Now, however, by implication $F_2 = 0$, so it is not possible to propagate through K_2. Therefore, we propagate through the pseudo-output SO_2. The 0 on SI_2 is justified by means of a 0 on J_1. That is justified by putting a 1 on primary input C_1.

A D now appears on the pseudo-input of time image 3. Assigning $G_3 = 0$ and $C_3 = 0$ places a D on the output of J_3. We set $B_3 = 1$ to justify the 0 from G_3 and then try to propagate the D on J_3 through K_3 by assigning $F_3 = 1$. This requires $D_3 = E_3 = 0$. We again find ourselves trying to set the faulted line to a 0. But this time we set it to $\overline{\text{D}}$, which causes D to appear on the output of F_3. Hence both inputs to K_3 are D and its output is $\overline{\text{D}}$. The final sequence of inputs is

	T_1	T_2	T_3
A	X	1	1
B	X	0	1
C	1	0	0

On the first time image, T_1 inputs A and B have X values. We assign values to these inputs as per the following rule: If the jth coordinate of the ith pattern is an X, then set it equal to the value of the jth coordinate on the first pattern number greater than i for which the jth coordinate has a non-X value. If no pattern greater than i has a value in the jth coordinate position, assign the most recent preceding value. If the jth coordinate is never assigned, then set it to the dominant value; that is, if the input feeds an AND gate set it to 0 and if it feeds an OR gate set it to 1. The objective is to minimize the number of input changes required for the test and hence minimize or eliminate races.

The reader may have noted that the cross-coupled NOR latch received input combination (1,1) in time image 1. According to its state table, this is an illegal input combination. Automatic test pattern generators occasionally assign combinations that are illegal or illogical when processing sequential circuits. It is one of the reasons why test patterns generated for sequential circuits must be verified through simulation.

5.3.3 The 9-Value ITG

When creating a test using ITG, it is sometimes the case that more constraints are imposed than are absolutely necessary. Consider again the circuit of Figure 5.4. We started by attempting to propagate a test through gate F. That would not work, so we propagated through G. If we look again at the problem and examine the immediate effects of propagating a test through gate F, we notice that the faulted circuit, because it produces a 0 on the upper input when $A = B = 1$, will produce a 1 on the output of K regardless of what value occurs on the lower input of K.

The D that was propagated to K implies that the upper input to K will be 1 in the fault-free circuit. Therefore the output of K for the unfaulted circuit depends on the value at its lower input. Since we want a sensitized signal on the output of K, the fault-free circuit must produce a 0 at the circuit output; therefore we want a 1 on the lower input to K.

A 1 can be obtained at the lower input to K by forcing J to produce a 1. This requires that both inputs to J be 0, which requires the output of H to be 0. Backing

TABLE 5.1 Symbols for Nine-Value ITG

Good	Faulted	ITG Symbol	D Symbol
0	0	0	0
0	X	G_0	—
0	1	S_0	$\overline{D}$
X	0	F_0	—
X	X	F_1	—
X	1	U	X
1	0	S_1	D
1	X	G_1	—
1	1	1	1

up one more step in the logic, we find that H is 0 if either the pseudo-input or G is 1. Gate G cannot be 1 because primary input B is 1. Therefore, a 1 must come from the pseudo-input. This is the point where we previously failed. The presence of the fault made it impossible to initialize the cross-coupled latch. Nevertheless, we will try again. However, this time we ignore the existence of the fault in the previous copy since we are only concerned with justifying a signal in the good circuit.

We create a previous time image and attempt to justify a 1 on its pseudo-output. A 1 can be obtained with $C = 0$ and $G = 1$, which requires $B = E = 1$, and implies $A = 0$. Therefore, a successful test is $I_1 = (1,0,0)$ and $I_2 = (1,1,0)$.

In order to distinguish between assignments required for faulted and unfaulted circuits, a nine-value algebra is used.[4] The definition of the nine values is shown in Table 5.1. The dashes correspond to unspecified values. The final column shows the corresponding values for the D-algorithm. It is readily seen that the D-algorithm symbols are a subset of the nine-value ITG symbols. Tables 5.2 through 5.4 define the AND, OR, and Invert operations on these signals.

TABLE 5.2 AND Operations on Nine Values

	0	G_0	S_0	F_0	U	G_1	S_1	F_1	1
0	0	0	0	0	0	0	0	0	0
G_0	0	G_0	G_0	0	G_0	G_0	0	G_0	G_0
S_0	0	G_0	S_0	0	G_0	G_0	0	S_0	S_0
F_0	0	0	0	F_0	F_0	F_0	F_0	F_0	F_0
U	0	G_0	G_0	F_0	U	U	F_0	U	U
G_1	0	G_0	G_0	F_0	U	G_1	S_1	U	G_1
S_1	0	0	0	F_0	F_0	S_1	S_1	S_0	S_1
F_1	0	G_0	S_0	F_0	U	U	S_0	F_1	F_1
1	0	G_0	S_0	F_0	U	G_1	S_1	F_1	1

TABLE 5.3 OR Operations on Nine Values

	0	G_0	S_0	F_0	U	G_1	S_1	F_1	1
0	0	G_0	S_0	F_0	U	G_1	S_1	F_1	1
G_0	G_0	G_0	S_0	U	U	G_1	G_1	F_1	1
S_0	S_0	S_0	S_0	F_1	F_1	1	1	F_1	1
F_0	F_0	U	F_1	F_0	U	G_1	S_1	F_1	1
U	U	U	F_1	U	U	G_1	G_1	F_1	1
G_1	G_1	G_1	1	G_1	G_1	G_1	G_1	1	1
S_1	S_1	G_1	1	S_1	G_1	G_1	S_1	1	1
F_1	F_1	F_1	F_1	F_1	F_1	1	1	F_1	1
1	1	1	1	1	1	1	1	1	1

TABLE 5.4 Invert Operations On Nine Values

X	0	G_0	S_0	F_0	U	G_1	S_1	F_1	1
Y	1	G_1	S_1	F_1	U	G_0	S_0	F_0	0

To illustrate the use of the tables, we employ the same circuit but start by assigning S_0 to the output of E_2 in Figure 5.5. The signal is propagated to the upper input of K_2, where, due to signal inversions, it becomes S_1. To propagate an S_1 through the NAND, we check the table for the AND gate. With S_1 on one of its inputs, a sensitized signal S_1 can be obtained at the output of the AND by placing either S_1, G_1, or a 1 on the other input. The inversion then causes the output of the NAND to become S_0. The signal G_1 is the least restrictive of the signals that can be placed on the other input since it imposes no requirements on the input for the faulted circuit.

Propagation requires a signal on the other input to F_2 that will not block the sensitized signal. From the table for the OR, we confirm that propagation through F_2 is

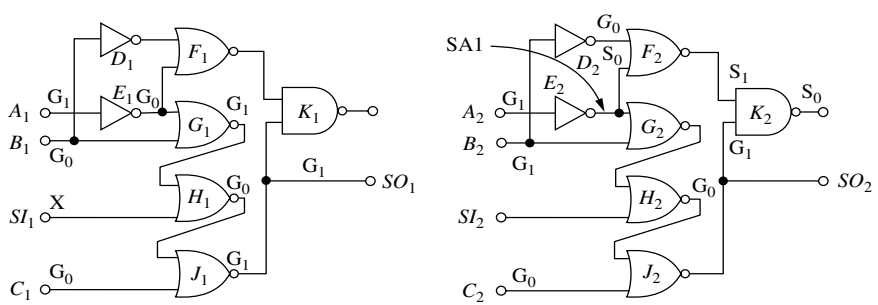


Figure 5.5 Test generation with the nine-value ITG.

successful with G_0 on the other input. That implies a G_1 on the input of gate D_2. Since the input to D_2 is a primary input, the signal is converted to 1. Justifying G_1 from J_2 requires G_0 from each of its inputs. Therefore, we need a G_0 from gate H_2, which implies a 1 at an input to H_2. The output of G_2 is 0 so the value G_1 must be obtained from the pseudo-input. We create a previous time image and require a G_1 from J_1. We then need G_0 from primary input C and also from H_1. That implies a G_1 from one of the inputs to H_1, which implies G_0 on both inputs to gate G_1. A G_0 from inverter E_1 is obtained by placing a G_1 on its input.

When justifying assignments, different values may be required on different paths emanating from a gate with fanout. These may or may not conflict, depending on the values required along the two paths. If one path requires G_1 and the other requires S_1, then both requirements can be satisfied with signal S_1. If one path requires G_1 and the other requires S_0, then there is a conflict because G_1 requires that the unfaulted circuit produce a logic 1 at the net and S_0 requires that the unfaulted circuit produce a logic 0.

5.3.4 The Critical Path

We have seen that, when attempting to develop a test for a sequential circuit, it is often not possible to reach a primary output in the present time frame (cf. Figure 5.2); fault effects must be propagated through flip-flops, into the next time image. But, when entering the next time frame, propagating the fault effect forward may require additional values from the previous time frame. Hence, it may become necessary to back up into the previous time frame in order to satisfy those additional values. This process of propagating, and then backing up into previous time frames, may occur repeatedly if a fault effect requires propagation through several future time frames. Resolving conflicts across time frames becomes a major problem. The critical path method described in Chapter 4 has sequential as well as combinational circuit processing capability. Because it always starts at a primary output and works back in time, it avoids this problem.

Its operation on a sequential circuit is described by means of an example, using the JK flip-flop of Figure 5.3. Recall that the critical path begins by assigning a value to an output. It then works its way back toward the input pins, creating a critical path along the way. Therefore, we start by assigning a 0 to the output of gate 13. This puts critical 1s on the inputs of gate 13, any one of which failing to the opposite state will cause an erroneous output.

Gate 11 is then selected. A 0 is assigned to gate 6 to force a 1 from gate 11. To make it critical we assign a 1 to gate 9. The assignment of a 0 to gate 6 forces assignment of 1s to input 3 and gate 12. Gate 14 is selected next. Since gate 13 is a 0 and gate 12 is a 1, we can create a critical 0 by assigning a 1 to input 5. The presence of a 0 on gate 13 also implies a 1 on the output of gate 8; hence gate 10 has a 0 on its output. To ensure that gate 9 has a 1, a 0 is assigned to gate 7. That in turn requires input 1 be assigned a 1.

Notice that the loop consisting of {13,14} has 1s on all predecessor inputs while the loop {9,10} is forced to its state by the 0 on gate 7. Since the inputs to loop {13,14} cannot force it to its state, the loop must be initialized to its state by a previous pattern. Therefore, the loop {13,14} becomes the initial objective of a preceding pattern. An assignment of 0 to input 5 and a 1 to inputs 1 and 3 forces the latch to the correct state.

One additional operation is performed here. The Clear input to gate 14 is made critical by reversing the values on the loop {13,14} in a previous third time image. The $\overline{\text{Preset}}$ is set to 0 and the $\overline{\text{Clear}}$ is set to 1. The complete input sequence then becomes

	T_1	T_2	T_3
1	0	1	1
2	X	X	1
3	1	1	1
4	X	X	X
5	1	0	1

The pattern at time T_1 resets the latch {13,14}. The pattern at time T_2 sets the latch; hence the 0 on input 5 at time T_2 is critical. Then, at time T_3, there is a critical path from input 3, through gates 6, 11, and 13. A failure on that path will cause the latch {13,14} to switch to the opposite state.

5.3.5 Extended Backtrace

The critical path is basically a justification operation, since its starting point is a primary output. Operating in this manner, it completely avoids the propagation operation, as well as the justification operations that may occur at each time-frame boundary. The extended backtrace (EBT)[5] bears some resemblance to the critical path. However, before backing up from a primary output, it selects a fault. Then, from that fault, a topological path (TP) is traced forward to an output. The TP may pass through sequential elements, indicating that several time frames are required to propagate the fault effect to an observable output. Along the way, other sequential subcircuits may need to be set up. This is illustrated in Figure 5.6.

In this hypothetical circuit, assume that the state machine has eight states and that input I controls the state transitions. Assume that net $L_2 = 1$ when in state S_8, $L_3 = 1$ when in state S_7, and $L_7 = 1$ when in S_6. Otherwise L_2, L_3, and L_7 equal 0. The comparator contains a counter, denoted B, and when the value in B equals the value on the A input port, net $L_1 = 0$, otherwise $L_1 = 1$. The goal is to create a test for the SA1 fault on net L_1.

One approach to solving this goal might be to begin by justifying the condition $A = B$ at the comparator. Once a match is obtained, the next clock pulse causes the

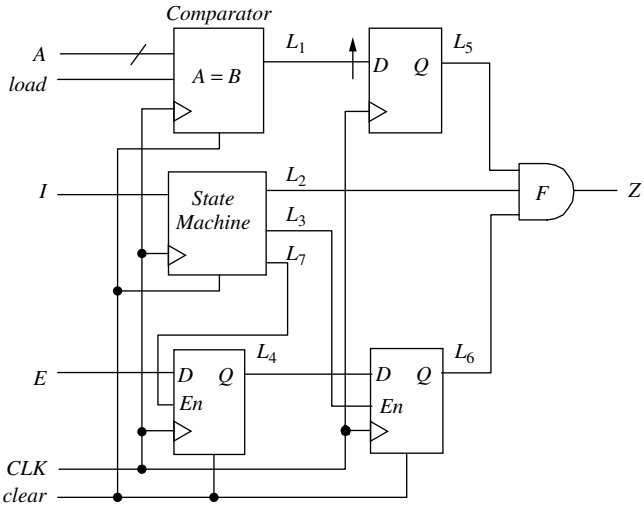


Figure 5.6 Aligning Sequential Circuits.

value 0 on L_1 to propagate through the flip-flop and reach AND gate F. To propagate through F it is necessary for nets L_2 and L_6 to be justified to 1. Should they be processed individually, or should they be processed in parallel? And should the vectors generated when processing L_2 and L_6 be positioned in the vector stream prior to, or after, those generated while justifying the comparator? The problem is complicated by the fact that L_6 not only depends on E, but also requires the state machine to transition through states S_6 and S_7, whereas L_2 requires the state machine to be in state S_8. The human observer can see that these are sequentially solvable, but the computer lacks intuition.

EBT begins by creating a TP to the output. The TP includes L_1, F, and Z. From the output Z, the requirement L_5, L_2, L_6 = (0,1,1) is imposed. This constitutes a current time frame (CTF) solution or vector. This CTF will often require a previous time frame (PTF) vector. The PTF is the complete set of assignments to flip-flops and primary inputs that satisfy the requirements for the CTF. Essentially, EBT is backing up along all paths in parallel, but with the proviso that the fault effect must propagate along the TP. Eventually, the goal is to reach a vector that does not rely on a PTF. At that point a self-initializing sequence exists that can test the fault. This last vector that is created is the first to be applied to the circuit.

EBT is simplified by the fact that forward propagation software is not required. However, the TP imposes requirements as it is traced forward, so during backtrace the TP requirements must be added to the requirements encountered during backtrace in order for the fault to become sensitized and eventually propagate forward to an output. Another advantage to EBT is the fact that vectors do not need to be inserted between vectors already created. Since processing always works backwards

in time, each PTF vector eventually becomes the CTF vector, and a new PTF is created, if necessary. Also, unlike critical path, EBT is fault oriented. This may permit shorter backtraces, since, for example, if a 1 is needed from a three-input NAND gate, the values (0,X,X) would be sufficient, whereas critical path requires (0,1,1). The trade-off, of course, is that there may be fewer fault detections per test vector sequence. In a complex sequential circuit, this may be a desirable trade-off.

5.3.6 Sequential Path Sensitization

The next system we look at is called the Sequential Path Sensitizer (SPS).[6] Its approach to sequential ATPG is to extend the D-notation into the time domain. The D and $\overline{\text{D}}$ of the combinational D-algorithm, together with their chaining rules, are subsumed into an expanded set of symbols and rules for creating chains that transcend time. All combinational logic in the cone (cf. Section 3.6.2) of a flip-flop or latch is gathered up and combined with the destination flip-flop to create a *super flip-flop*. Similarly, all combinational logic in the cone of a primary output is treated as a *super output block*. State transition properties, including *extended D-cubes*, for these super flip-flops are derived in terms of the behaviors of latches and flip-flops.

In another departure from conventional practice, SPS does not explicitly model faults. Rather, it sensitizes paths from primary inputs to primary outputs via sequences of input vectors and then propagates 0 and 1 along the path.[7] If an incorrect response occurs at an output during testing, the defect lies either along the sensitized path or on some attendant path used to sensitize the critical path. Path intersection can be used to isolate the source of the erroneous response.

We begin by considering the behavior of a negative edge triggered JK flip-flop with output F and inputs J, K, R, S, and C, where the S and R inputs are active high. The JK flip-flop is capable of four distinct activities: Set, Reset, Toggle, and At-Rest, denoted by the symbols σ, ρ, τ, and α. The following equations express these actions:

Set: $$\sigma = S \cdot \bar{R} \cdot \overline{(J \cdot \bar{K} \cdot C/\bar{C})} + J \cdot \bar{K} \cdot \bar{S} \cdot \bar{R} \cdot C/\bar{C} \tag{5.1}$$

Reset: $$\rho = \bar{S} \cdot R \cdot \overline{(\bar{J} \cdot K \cdot C/\bar{C})} + \bar{J} \cdot K \cdot \bar{S} \cdot \bar{R} \cdot C/\bar{C} \tag{5.2}$$

Toggle: $$\tau = J \cdot K \cdot \bar{S} \cdot \bar{R} \cdot C/\bar{C} \tag{5.3}$$

At Rest: $$\alpha = \bar{J} \cdot \bar{K} \cdot \bar{S} \cdot \bar{R} + \bar{S} \cdot \bar{R} \cdot \overline{C/\bar{C}} \tag{5.4}$$

In these equations, $C/\bar{C}$ denotes a true-to-false clock transition and $\overline{C/\bar{C}}$ denotes absence of the true-to-false transition. A complete set of state transitions can be expressed in terms of the preceding four equations. These yield

$$F(i+1)/1 = \sigma + \tau\bar{F}(i) + \alpha F(i) \tag{5.5}$$

$$F(i+1)/0 = \rho + \tau F(i) + \alpha\bar{F}(i) \tag{5.6}$$

TABLE 5.5 Some D-Cubes

F	S	R	J	K	C	Initial State F	Equation/Term
D	D	0	X	X	0	0	5.1/1
D	D	0	0	0	X	0	5.1/1
D	D	0	X	X	1	0	5.1/1
D	0	0	D	0	1/0	0	5.1/1
D	0	0	1	0	D/0	0	5.1/2
D	0	0	1	1	D/0	0	5.1/2
$\overline{D}$	0	D	X	X	0	1	5.3
$\overline{D}$	0	D	0	0	X	1	5.2/1
$\overline{D}$	0	D	X	X	1	1	5.2/1
$\overline{D}$	0	0	0	$\overline{D}$	1/0	1	5.2/2
$\overline{D}$	0	0	0	1	D/0	1	5.2/2
$\overline{D}$	0	0	1	1	D/0	1	5.3

where $F(i)/1$ indicates that F is true at time i and $F(i)/0$ indicates that F is false at time i. Equation (5.5) states that a true output occurs at time $i + 1$ if a set is performed, or if the flip-flop is toggled when it is originally in the false state, or if it is true and is left at rest. Equation (5.6) is interpreted similarly. From these equations, primitive D-cubes can be derived that are then used to define local transition conditions for the super flip-flops. They constitute a covering set of cubes for the σ, ρ, τ, and α and state control equations. Some of the D-cubes are listed in Table 5.5.

Corresponding to the D-cubes listed in the table is a set of inhibit D-cubes that can be obtained by complementing all of the D and $\overline{D}$ terms. The final column in the table indicates the derivation of the D-cube. For example, the first D-cube was derived from the first term of Eq. (5.1). The interpretation of each entry is similar to that of the D-cubes of the D-algorithm. The first D-cube states that with Clock and Reset at 0, and flip-flop output F at 0, the output F is sensitive to a D on the Set input. The coordinates within each cube are grouped in terms of output variables, internal variables, and controllable input variables. The cubes for a given condition are arranged in hierarchical order corresponding inversely to the number of non-X state memory variable coordinates in the cube required to facilitate generation of initializing sequences. In all, four distinct activities are defined for SPS:

1. Identify super flip-flops and super output blocks. Determine D-cubes for each of these super logic blocks.
2. Trace super logic block D-cubes to define sequential D-chains that define sequential circuit propagation paths.
3. Determine an exercise sequence for each sequential logic D-chain.
4. Determine an initialization sequence for each sequential logic D-chain.

In the first step, after defining the super logic blocks as described earlier and developing D-cubes for the basic memory elements, this information is used to

develop D-cubes for the super logic blocks by extending the basic memory element D-cubes through the preceding combinational logic.

In the second step, beginning with a super logic block D-cube that generates an observable circuit output, proceed as in the D-algorithm to chain D-cubes back to inputs. During this justification phase, other super flip-flops may be reached that are inputs to the one being processed. These super flip-flops are chained as in the D-algorithm by means of an extended set of symbols to permit computation of state transitions. The extended symbols and their intersection rules are given in Table 5.6. An explanation of the symbols follows the table.

Note that in the explanation some symbols are identified as input symbols and some are identified as output symbols. The output symbols identify possible states of super flip-flops that correspond to possible states of the latch or JK flip-flop from which the super flip-flop was derived. Therefore, the outputs of these super flip-flops are expressed in terms of true and false final states, toggles, and at-rest conditions. When using Table 5.6 to intersect an input value with an output value, the result provided by the table is a flip-flop output value that is compatible with input requirements on the element(s) driven by that flip-flop. For example, if element inputs connected to a net require a logic 1 in a present time frame, then that

TABLE 5.6 Intersection Table

	$\overline{D}$	D		D/0	$\overline{D}$/1								
	0	1	X	1/0	0/1	d	$\overline{d}$	T	$\overline{T}$	t	$\bar{t}$	A	$\overline{A}$
$\overline{D}$, 0	0	*	0	*	0/1	*	$\overline{d}$	*	$\overline{T}$	$\overline{A}$	$\overline{T}$	*	$\overline{A}$
D, 1	*	1	1	1/0	*	d	*	T	*	T	A	A	*
X	0	1	X	1/0	0/1	d	$\overline{d}$	T	$\overline{T}$	t	$\bar{t}$	A	$\overline{A}$
D/0, 1/0	*	1/0	0/1	1/0	*	*	*	T	*	T	*	*	*
$\overline{D}$/1, 0/1	0/1	*	0/1	*	0/1	*	*	*	$\overline{T}$	*	$\overline{T}$	*	*
d	*	d	d	*	*	d	*	*	$\overline{T}$	*	$\bar{t}$	A	*
$\overline{d}$	$\overline{d}$	*	$\overline{d}$	*	*	*	$\overline{d}$	T	*	t	*	*	$\overline{A}$
T	*	T	T	T	*	*	T	—	—	—	—	—	—
$\overline{T}$	$\overline{T}$	*	$\overline{T}$	*	$\overline{T}$	$\overline{T}$	*	—	—	—	—	—	—
t	$\overline{A}$	T	t	T	*	*	t	—	—	—	—	—	—
$\bar{t}$	$\overline{T}$	A	$\bar{t}$	*	$\overline{T}$	$\bar{t}$	*	—	—	—	—	—	—
A	*	A	A	*	*	A	*	—	—	—	—	—	—
$\overline{A}$	$\overline{A}$	*	$\overline{A}$	*	*	*	$\overline{A}$	—	—	—	—	—	—

Inputs	Outputs
1 = true state	$\bar{t}$ = true final state
0 = false state	t = false final state
X = don't care	$\overline{T}$ = 0/1 toggle
1/0 = true-to-false transition	T = 1/0 toggle
0/1 = false-to-true transition	A = true at rest
$\overline{D}$, D, D/0, $\overline{D}$/1 = D-states	$\overline{A}$ = false at rest
d, $\overline{d}$ = asynchronous D-inputs	* = prohitited state

value can be justified by a flip-flop that is true at rest, *A*, or one that is presently true but which will toggle to false on the next time frame, either *t* or *T*. The symbols *t* and *T* have identical meaning during the exercising sequence: They differ slightly during the initializing sequence, as will be explained later. The dashes indicate impossible conditions and the asterisks correspond to conflicting choices, as in the original D-algorithm.

When intersecting D-cubes, the following rules must be followed:

1. No latch or flip-flop output may be left with a 1/0, 0/1, D/1 or $\overline{D}$/0 state.
2. There must be no d or $\overline{d}$ terms left on the latch or flip-flop coordinates of a resultant cube.
3. Cubes that are asynchronously coupled via unclocked inputs must be intersected in the same time frame.

If a toggle state occurs, additional cubes must be combined with the original cube in order to completely define that step of the sequence. Cubes that are coupled by means of a d or $\overline{d}$ or by means of unclocked inputs must be combined via intersection.

The circuit in Figure 5.7 will be used to illustrate the sequential path sensitizer. Cubes are chained from the output back toward inputs, and these are used to create an initializing and exercising sequence for the propagation path.

We begin by identifying the super flip-flops and the super output block. The super output consists of a single AND gate labeled block *Z*. There are two JK flip-flops and a Set–Reset (S–R) latch. The JK flip-flop behavior is described by Eqs. (5.1)–(5.6). The S–R latch is at rest when both inputs are low. It is set (output high) or reset (output low) when the corresponding input is high. The S–R latch and flip-flop *Y* have no combinational logic preceding them. The JK flip-flop labeled *V* is preceded by an OR gate, two inverters, and two AND gates. These gates and flip-flop *V* are bundled together and processed as a single super

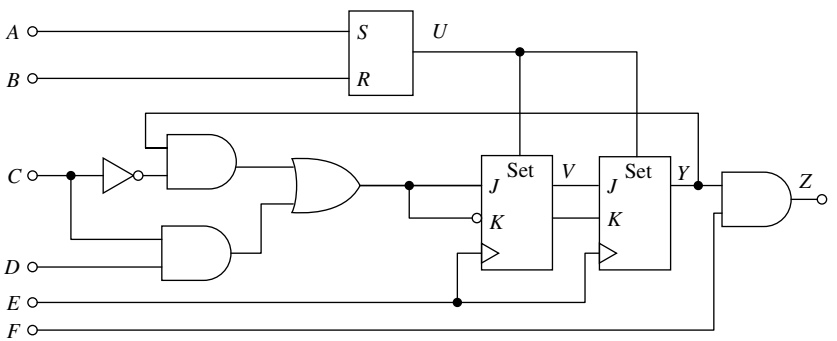


Figure 5.7 Circuit for sequential path sensitization.

TABLE 5.7 Super Flip-Flop Cubes

Z	*U*	*V*	*Y*	*A*	*B*	*C*	*D*	*E*	*F*	Cube name
$\bar{t}$	X	X	d	X	X	X	X	X	1	$Z\sigma_1$
$\bar{t}$	X	X	1	X	X	X	X	X	d	$Z\sigma_2$
t	X	X	$\overline{d}$	X	X	X	X	X	1	$\overline{Z}\rho_1$
t	X	X	1	X	X	X	X	X	$\overline{d}$	$\overline{Z}\rho_2$
X	d	X	$\bar{t}$	X	X	X	X	X	X	$Y\sigma_1$
X	0	D	t	X	X	X	X	1/0	X	$Y\sigma_2$
X	0	$\overline{D}$	t	X	X	X	X	1/0	X	$\overline{Y}\rho$
X	0	X	A	X	X	X	X	0	X	$Y\alpha$
X	0	X	$\overline{A}$	X	X	X	X	0	X	$\overline{Y}\alpha$
X	d	$\bar{t}$	X	X	X	X	X	X	X	$V\sigma_1$
X	0	$\bar{t}$	X	X	X	1	D	1/0	X	$V\sigma_2$
X	0	$\bar{t}$	X	X	X	1	1	D/0	X	$V\sigma_3$
X	0	$\bar{t}$	D	X	X	0	X	1/0	X	$V\sigma_4$
X	0	$\bar{t}$	1	X	X	$\overline{D}$	0	1/0	X	$V\sigma_5$
X	0	A	X	X	X	X	X	0	X	$V\alpha$
X	0	t	X	X	X	1	$\overline{D}$	1/0	X	$\overline{V}\rho_1$
X	0	t	$\overline{D}$	X	X	0	X	1/0	X	$\overline{V}\rho_2$
X	0	t	0	X	X	$\overline{D}$	1	1/0	X	$\overline{V}\rho_3$
X	0	$\overline{A}$	X	X	X	X	X	0	X	$\overline{V}\alpha$
X	$\bar{t}$	X	X	d	0	X	X	X	X	$U\sigma$
X	A	X	X	0	0	X	X	X	X	$U\alpha$
X	t	X	X	0	d	X	X	X	X	$\overline{U}\rho$
X	$\overline{A}$	X	X	0	0	X	X	X	X	$\overline{U}\alpha$

flip-flop. The next step is to create D-cubes for the four super flip-flops *U*, *V*, *Y*, and *Z*. These cubes are contained in Table 5.7 and are assigned names to facilitate the description that follows.

The cube name consists of the letter *U*, *V*, *Y*, or *Z* originally assigned to the super flip-flop, complemented if necessary, followed by one of the symbols σ, ρ, τ, or α to indicate whether the action is a Set, Reset, Toggle, or At-Rest. If more than one entry exists for an action, they are numbered.

Having created D-cubes for the super output block and the super flip-flops, sequential paths from the outputs to the inputs are identified in order to construct an exercising sequence. If the cube $Z\sigma_1$ is selected, corresponding to a true state on the output *Z*, we see that it specifies a d on flip-flop *Y*, which must now be justified.

The d is justified by going across the top of Intersection Table 5.6 until reaching the column labeled d. In that column there appear to be six possible choices. However, only three of the entries in that column, $\bar{t}$, $\overline{T}$, and A, can be obtained from the output of a super flip-flop. Going across those rows to the left, we see that signals $\bar{t}$, $\overline{T}$, and A can be created by intersection with $\bar{t}$, $\overline{T}$, and A. We then go to the set of D-cubes for *Y* in Table 5.7 and search for one that produces $\bar{t}$, $\overline{T}$, or A without causing a

conflict. For purposes of illustration we select $Y\sigma_2$. It requires a D from input V and a 0 from input U.

Table 5.6 is used to justify the D. The column with header D reveals that a D occurs at the input to Y if V is true while at rest, A, or if it is presently true but toggles false, T, at the next time frame. Since no cubes exist in Table 5.7 with a T on the output of V, we check entries from Table 5.6 with A and find, by going across to the left, that they result from intersection with either an A or $\bar{t}$ on the output of V. From the D-cubes for V in Table 5.7, $V\sigma_4$ is selected. Finally, in similar fashion, a 0 is justified on U by means of cube $\overline{U}\alpha$.

Four cubes have now been identified that extend a sensitized path back from output Z to primary inputs and other elements. Before continuing, we point out that the sensitized path extends through both logic and time, since the cubes impose switching conditions as well as logic values. As a result, intersections are more complex and require attention to more detail than is the case with the D-algorithm. Some cubes must be intersected in the same time frame, and others, linked by synchronous switching conditions, are used to satisfy conditions required in the preceding time frame.

Consider the first D-cube selected, $Z\sigma_1$. It creates a $\bar{t}$ on the output of Z by assigning a 1 and a d to the inputs of the AND gate. The 1 is satisfied by assigning a 1 to input F. The d, which is an asynchronous D, must be justified in the present time frame. This is accomplished by intersecting $Z\sigma_1$ with the second cube previously selected, $Y\sigma_2$. Performing the intersection according to the rules in Table 5.6, we obtain the following:

$\bar{t}$	X	X	d	X	X	X	X	X	1	$Z\sigma_1$
X	0	D	$\bar{t}$	X	X	X	X	1/0	X	$Y\sigma_2$
$\bar{t}$	0	D	$\bar{t}$	X	X	X	X	1/0	1	

The resultant cube applies a 0 to the Set input of flip-flop Y. The fourth cube previously selected, $\overline{U}\alpha$, which was chosen to justify the 0 on the Set input, is asynchronously coupled to Y via the unclocked Set input. Therefore, according to the intersection rules, it must be intersected with the previous result.

$\bar{t}$	0	D	$\bar{t}$	X	X	X	X	1/0	1	
X	$\overline{A}$	X	X	0	0	X	X	X	X	$\overline{U}\alpha$
$\bar{t}$	$\overline{A}$	D	$\bar{t}$	0	0	X	X	1/0	1	

The remaining cube, $V\sigma_4$, was selected to justify a D on the input to Y. Since the input is synchronized to the clock, the cube $V\sigma_4$ becomes part of the preceding time frame. Values on Z, U, V, and Y for this resultant cube are interpreted by using the legends at the bottom of Table 5.6. Super blocks Z, U and Y have both a final value

and a switching action specified. During an exercising sequence the $\bar{t}$ denotes a transition on the outputs of Z and Y from a present state of 0 to a final state of 1. The $\overline{A}$ on $\overline{U}$ denotes a super flip-flop that is false at rest; that is, its final value is false and, furthermore, it did not change. Therefore, the Set input to Y is inactive. Super flip-flop V has a D, which is an input value; therefore no final value is specified for that super flip-flop.

The interpretation, then, of the resultant cube is that there is an output of 1, 0, X, 1 at time $n + 1$ from the four super blocks. At time n the circuit requires values 0, 0, 1, 0 on the outputs of the super blocks and values *A, B, C, D, E, F* = (0, 0, X, X, 1/0, 1) on the primary inputs. Note that the clock value is specified as 1/0 and is regarded as a single stimulus, although in fact it requires two time images.

The values (*Z, U, V, Y*) = (0, 0, 1, 0) required on the super blocks at time n must now be justified. The original third cube, $V\sigma_4$, which was selected to justify a D at the input to *V*, puts a $\bar{t}$ on the output of *V* and requires a 0 on the input driven by *U*. Its combinational logic inputs require a 0 on input *C* and a D on the input from super flip-flop *Y*. The $\bar{t}$ represents a true final state on *V* and therefore satisfies the requirement imposed by the previously created pattern. However, we still need 0s on the other super flip-flops. We must justify these values without conflicting with values of the cube $V\sigma_4$.

There is already an apparent conflict. The cube requires a D on *Y*, and the previously created cube requires a 0 on *Y*. However, the D is an input to the super flip-flop at time $n - 1$ as specified by the cube $V\sigma_4$. The 0 is an output requirement at time n and the cube $V\sigma_4$ specifies that flip-flop V is to perform a toggle. The apparent problem is caused by the fact that a loop exists. We attempt to justify the 0 required on *U*. The cube $\overline{U}\rho$ will justify the 0. We then select $\overline{Z}\rho_1$ to get a 0 on *Z*, and we select $\overline{Y}\rho$ to get a 0 on *Y*. The intersection of these cubes yields the following:

t	X	X	$\bar{d}$	X	X	X	X	X	1	$\overline{Z}\rho_1$
X	0	$\overline{D}$	t	X	X	X	X	1/0	X	$Y\sigma_2$
X	t	X	X	0	d	X	X	X	X	$\overline{U}\rho$
X	0	$\bar{t}$	D	X	X	0	X	1/0	X	$V\sigma_4$
t	$\overline{A}$	$\overline{T}$	T	0	d	0	X	1/0	1	

All columns except column 4, corresponding to super flip-flop *Y*, follow directly from the intersection table. As mentioned, the fourth column requires a $\bar{d}$ output from *Y* and a D input. In addition, the cube $Y\sigma_2$ requires a 1/0 toggle. Therefore, we intersect a D and t to get T and then intersect T with $\bar{d}$ to again get a T. The exercising sequence is now complete. The values t, $\overline{A}$, $\overline{T}$, T satisfy the requirements for 0, 0, 1, 0 that we set out to obtain, but they in turn impose initial conditions of 1, 0, 0, 1. We therefore must create an initialization sequence by continuing to justify backward in time until we eventually reach a point in which

all of the super blocks have X states. To satisfy the assignments 1, 0, 0, 1, we intersect the following:

$\bar{t}$	X	X	d	X	X	X	X	X	1	$Z\sigma_1$
X	t	X	X	0	d	X	X	X	X	$\overline{U}\rho$
X	0	t	X	X	X	1	$\overline{D}$	1/0	X	$\overline{V}\rho_1$
X	0	D	$\bar{t}$	X	X	X	X	1/0	X	$Y\sigma_2$
$\bar{t}$	$\overline{A}$	T	$\bar{t}$	0	d	1	$\overline{D}$	1/0	1	

During creation of the initialization sequence, we are aided by an additional observation. The $\bar{t}$, which implied a true final state and a false start state while building the exercising sequence, still implies a true final state but implies an *x* state while constructing the initializing sequence. Therefore the values $\bar{t}$, $\overline{A}$, T, $\bar{t}$ on the super blocks satisfy the 1,0,0,1 requirement and also imply a previous state of X, 0, 1, X on the super block outputs. Thus, two of the super blocks can be ignored.

To get the previous state in which $U = 0$ and $V = 1$, we intersect:

X	$\overline{A}$	X	X	0	0	X	X	X	X	$\overline{U}\alpha$
X	0	$\bar{t}$	X	X	X	1	D	1/0	X	$V\sigma_2$
X	$\overline{A}$	$\bar{t}$	X	0	0	1	D	1/0	X	

Again, the $\bar{t}$ satisfies the requirement for V = 1 and specifies a previous don't care state. Since we are constructing an initializing sequence at this point, rather than an exercising sequence, the D is ignored; that is, it is treated as a logic 1. A 0 is now required on the output of super flip-flop *U*. The D-cube $\overline{U}\rho$ is used, which puts a t on the output of the flip-flop, hence a 0 preceded by a don't care state. The inputs for that cube are 0 and d. The d is again treated as a 1 because this is the initializing sequence. The task is done; we now go back and reconstruct the entire sequence. We get:

n	*Z*	*U*	*V*	*Y*	*A*	*B*	*C*	*D*	*E*	*F*
1	X	X	X	X	0	1	X	X	X	X
2	X	0	X	X	0	0	1	1	1/0	X
3	X	0	1	X	0	1	1	0	1/0	1
4	1	0	0	1	0	1	0	X	1/0	1
5	0	0	1	0	0	0	X	X	1/0	1
6	1	0	X	1						

5.4 SEQUENTIAL LOGIC TEST COMPLEXITY

A general solution to the test problem for sequential logic has proven elusive. Recall that several algorithms exist that can find a test for any fault in a combinational circuit,

if a test exists, given only a list of the logic elements used in the circuit and their interconnections. No comparable theoretical basis for sequential circuits exists under the same set of conditions.

5.4.1 Acyclic Sequential Circuits

The analysis of sequential circuits begins with the circuit of Figure 5.8. Although it is sequential, it is loop-free, or acyclic. There is no feedback, apart from that which exists inside the flip-flops. In fact, the memory devices need not be flip-flops, the circuit could be implemented with delays or buffers to obtain the required delay. The circuit would not behave exactly the same as a circuit with clocked flip-flops, since flip-flops can hold a value for an indefinite period if the clock is halted, whereas signals propagate unimpeded through delay lines. However, with delay lines equalling the clock period, it would be impossible for an observer strobing the outputs to determine if the circuit were implemented with delay lines or clocked flip-flops.

If the circuit is made up of delay lines, then for many of the faults the circuit could be considered to be purely combinational logic. The signal at the output fluctuates for a while but eventually stabilizes and remains constant as long as the inputs are held constant. If a tester connected to the output samples the response at a sufficiently late time relative to the total propagation time through the circuit, the delay lines would have no more effect than wires with zero delay and could therefore be completely ignored.

If the delays are flip-flops, how much does the analysis change? Suppose the goal is to create a test for an SA1 fault on the top input to gate B_4. A test for the SA1 fault can be obtained by setting $I_1 = 0$, $FF_2 = \mathrm{X}$ and $FF_3 = 1$. If FF_4 represents time image n, then a 1 is required on primary input I_6 in time image $n - 1$ in order to justify the 1 on FF_3 in time image n. Propagation through FF_5 in time image $n + 1$ is achieved by requiring $FF_7 = 1$. That can be justified by setting $I_5 = 1$ in time image n and $I_4 = 1$ in time image $n - 1$. The entire sequence becomes

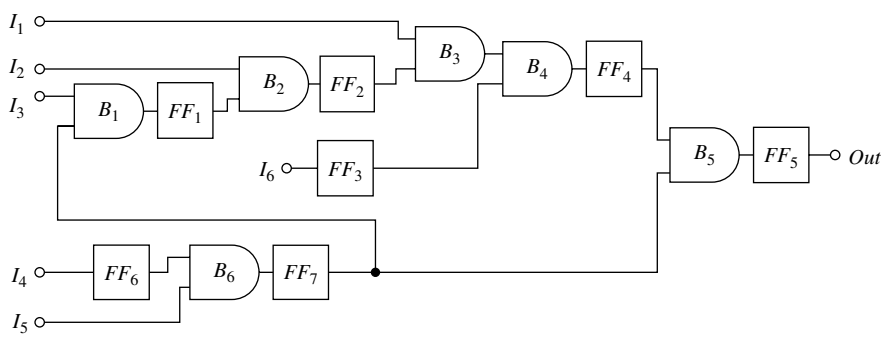


Figure 5.8 An acyclic sequential circuit.

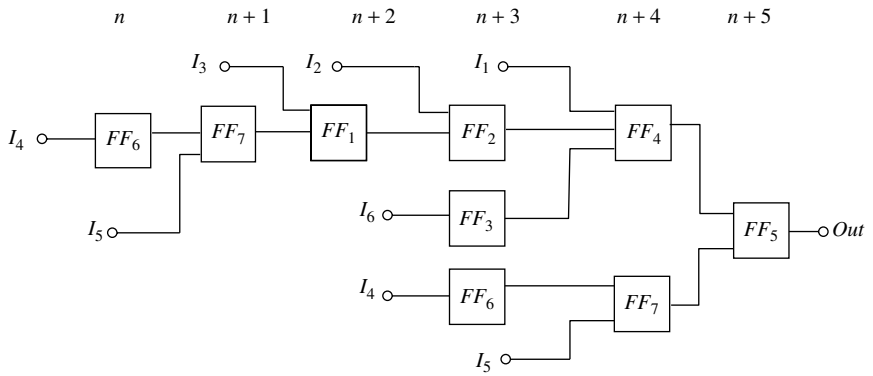


Figure 5.9 The acyclic rank-ordered circuit.

Time	I_1	I_2	I_3	I_4	I_5	I_6	Out
$n-1$	X	X	X	1	X	1	X
n	0	X	X	X	1	X	X
$n+1$	X	X	X	X	X	X	X
$n+2$							$\overline{\text{D}}$

To summarize, a fault is sensitized in time image n, and assignments are justified backward in time to image $n-1$ and are propagated forward in time to image $n+1$. The result finally appears at an observable output in time image $n+2$. Of interest here is the fact that the test pattern could almost as easily have been generated by a combinational ATPG. The circuit has been redrawn as an S-graph in Figure 5.9, where the nodes in the graph are the original flip-flops. The logic gates have been left out but the connections between the nodes represent paths through the original combinational logic. The nodes have been rank-ordered in time, with the time images indicated at the top of Figure 5.9. Because FF_7 fans out, it appears twice, as does its source FF_6.

In order to test the same fault in the redrawn circuit, the flip-flops can be ignored while computing input stimuli and the rank-ordered circuit can be used to determine the time images in which stimuli must occur. For test purposes, the complexity of this circuit is comparable to that of a combinational circuit. Since the number of test patterns for a combinational circuit with n inputs is upper bounded by 2^n, the number of test patterns for this pseudo-combinational circuit is upper-bounded by $k \cdot 2^n$, where k is circuit depth; that is, k is the maximum number of flip-flops in any path between any input and any output.

Example A test will be created for the bottom input of B_4 SA1. The input stimuli are

I_1	I_2	I_3	I_4	I_5	I_6
1	1	1	1/1	1/1	0

The double assignments for I_4 and I_5 represent values at different times due to fanout. If destination flip-flops exist in different time images, we can permit what would normally be conflicting assignments. If the fanout is to two or more destination flip-flops, all of which exist in the same time image, then the assignments must not conflict. From the rank-ordered circuit it is evident that the values must occur in the following time images:

Out	I_1	I_2	I_3	I_4	I_5	I_6	Time
X	X	X	X	1	X	X	n
X	X	X	X	X	1	X	$n+1$
X	X	X	1	X	X	X	$n+2$
X	X	1	X	1	X	0	$n+3$
X	1	X	X	X	1	X	$n+4$
X	X	X	X	X	X	X	$n+5$
$\overline{\text{D}}$							$n+6$

The previously generated test sequence can be shifted three units forward in time and merged with the second test sequence to give

Out	I_1	I_2	I_3	I_4	I_5	I_6	Time
X	X	X	X	1	X	X	n
X	X	X	X	X	1	X	$n+1$
X	X	X	1	1	X	1	$n+2$
X	0	1	X	1	1	1	$n+3$
X	1	X	X	X	1	X	$n+4$
$\overline{\text{D}}$	X	X	X	X	X	X	$n+5$
$\overline{\text{D}}$							$n+6$

■■

5.4.2 The Balanced Acyclic Circuit

The concept of using a combinational ATPG for the circuit of Figure 5.8 breaks down for some of the faults. For example, an SA0 on the top input to B_6, driven by FF_6, cannot be tested in this way because the fault requires a 0 for sensitization and a 1 for propagation. The circuit is said to be unbalanced because there are two fanout paths from FF_7 to the output and there are a different number of flip-flops in each of the fanout paths.

When every path between any two nodes in an acyclic sequential circuit has the same number of flip-flops, it is called a *balanced acyclic sequential circuit*. The *sequential depth* d_{max} of the balanced circuit is the number of nodes or vertices on the longest path in the S-graph. Given a balanced circuit, the sequential elements in

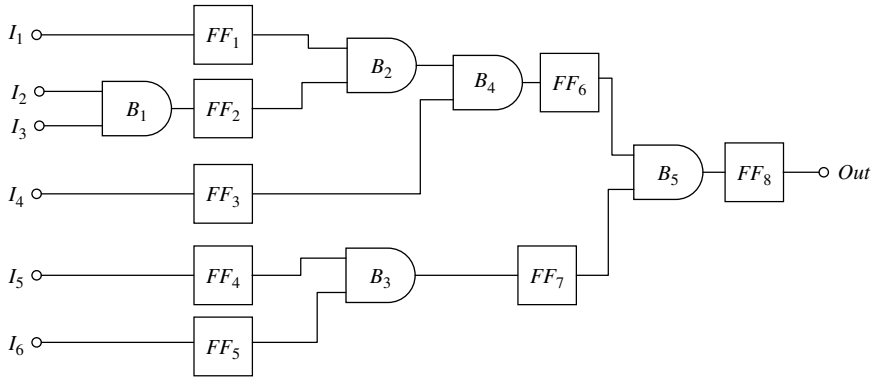


Figure 5.10 A strongly balanced circuit.

the model can be replaced by wires or buffers. Vectors can then be generated for faults in the resulting circuit model using a combinational ATPG. The vector thus generated is applied to the circuit for a duration of $d_{max} + 1$ clock cycles.[8]

An *internally balanced acyclic sequential circuit* is one in which all node pairs except those involving primary inputs are balanced.[9] Like the balanced sequential circuit, the internally balanced circuit can be converted to combinational form by replacing all flip-flops with wires or buffers. However, one additional modification to the circuit model is required: The primary inputs that are unbalanced are split and represented by additional primary inputs so that the resulting circuit is balanced. Then, the combinational ATPG can be used to create a test pattern. Each test pattern is replicated $d_{max} + 1$ times. The logic bits on the replicated counterpart I_j' to the original input I_j must be inserted into the bitstream for input I_j at the appropriate time.

Another distinction can be made with respect to balanced circuits. A *strongly balanced acyclic circuit* is balanced and, in addition, all paths from any given node in the circuit to the primary inputs driving its cone have the same sequential depth.[10] This is illustrated in Figure 5.10. A backtrace from *Out* to any primary input encounters three flip-flops. For test purposes, the model can be altered such that the flip-flops are converted to buffers. Then, test vectors for individual faults can be generated by a combinational ATPG. These are then stacked and clocked through the actual circuit on successive clock periods. The last vector, applied to the inputs at time n, will cause a response at *Out* during time $n + 3$.

A hierarchy of circuit types, based on sequential constraints, is represented in Figure 5.11 (combinational circuits are most constrained). A general sequential circuit can be converted to acyclic sequential by means of scan flip-flops (cf. Chapter 8). The flip-flops to be scanned can be chosen using a variant of the loop-cutting algorithm described in Section 5.3.2. Given an acyclic circuit, it has been shown that a balanced model of the circuit can be created for ATPG purposes. Each

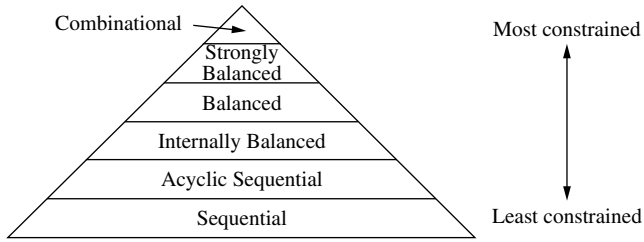


Figure 5.11 Classification based on sequential constraints.

vector created by the combinational ATPG is then transformed into a test sequence for the actual circuit.[11] It is reported that this approach reduces the ATPG time by an order of magnitude while producing vector lengths comparable to those obtained by sequential ATPGs.

5.4.3 The General Sequential Circuit

Consider what happens when we make one alteration to the circuit in Figure 5.8. Input I_5 is eliminated and a connection is added from the output of B_5 to the input of B_6. With this one slight change the entire nature of the problem has changed and the complexity of the problem that we are trying to solve has been compounded by orders of magnitude. In the original circuit the output was never dependent on inputs beyond six time frames. Furthermore, no flip-flop was ever dependent on a previous state generated in part by that same flip-flop.

That has changed. The four flip-flops FF_1, FF_2, FF_4, and FF_7 constitute a state machine of 16 states in which the present state may be dependent on inputs that occurred at any arbitrary time in the past. This can be better illustrated with the state transition graph of Figure 5.12. If we start in state S_1 the sequence 1011111... takes

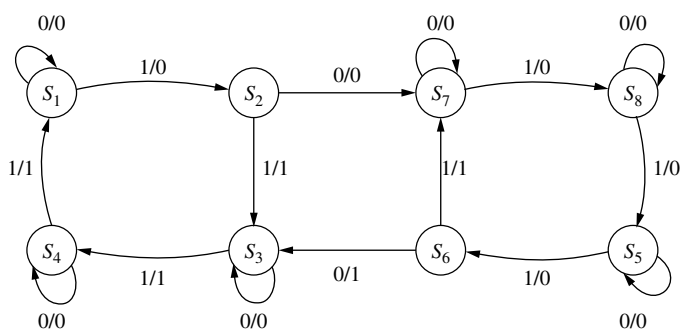


Figure 5.12 State transition graph.

us to $S_2\{S_7, S_8, S_5, S_6\}^*$, where the braces and asterisk denote an arbitrary number of repetitions of the four states in braces. From the almost identical sequence 11011111..., we get the state sequence S_2, $S_3\{S_3,S_4,S_1,S_2\}^*$. The corresponding output sequences are $0,0\{0,0,0,1\}^*$ and $0,1,0\{1,1,0,1\}^*$, a significant difference in output response that will continue as long as the input consists of a string of 1s. In a circuit with no feedback external to the flip-flops the output sequences will coincide within k time images where k again represents the depth of the circuit.

How much effect does that feedback line have on the testability of the circuit? We will compute an upper bound on the number of test patterns required to test a state machine in which the present state is dependent on an input sequence of indeterminate length—that is, one in which present state of the memory cells is functionally dependent upon a previous state of those same memory cells.

Given a state machine with n inputs and M states, $2^{m-1} < M < 2^m$, and its corresponding state table with M rows, one for each state, and 2^n columns, one for each input combination, there could be as many as 2^n unique transitions out of each state. Hence, there could be as many as $M \cdot 2^n$, or approximately 2^{m+n}, transitions that must be verified. Given that we are presently in state S_i, and we want to verify a transition from state S_j to state S_k, it may require $M - 1$ transitions to get from S_i to S_j before we can even attempt to verify the transition $S_j \rightarrow S_k$. Thus, the number of test vectors required to test the state machine is upper bounded by 2^{2m+n}, and that assumes we can observe the present state without requiring any further state transitions.

The argument was derived from a state table, but is there a physical realization requiring such a large number of tests? A realization can, in fact, be constructed directly from the state table. The circuit is implemented with m flip-flops, the outputs of which are used to control m multiplexers, one for each flip-flop. Each multiplexer has M inputs, one for each row of the state table. Each multiplexer input is connected to the output of another multiplexer that has 2^n inputs, one corresponding to each column of the state table. The inputs to this previous bank of multiplexers are fixed at 1 and 0 and are binary m-tuples corresponding to the state assignments and the next states in the state table. In effecting state transitions, the multiplexers connected directly to the flip-flops select the row of the state table and the preceding set of multiplexers, under control of the input signal, select the column of the state table, thus the next state is selected by this configuration of multiplexers.

In this implementation $M \cdot 2^n$ m-tuples must be verified, one for each entry in the state table. From the structure it can be seen that checking a given path could require as many as $M - 1$ transitions of the state machine to get the correct selection on the first bank of multiplexers. Consequently, the number of test patterns required to test this implementation is upper bounded by 2^{2m+n}. This is not a practical way to design a state machine, but it is necessary to consider worst-case examples when establishing bounds. Of more significance, the implementation serves to illustrate the dramatic change in the nature of the problem caused by the presence of feedback lines.

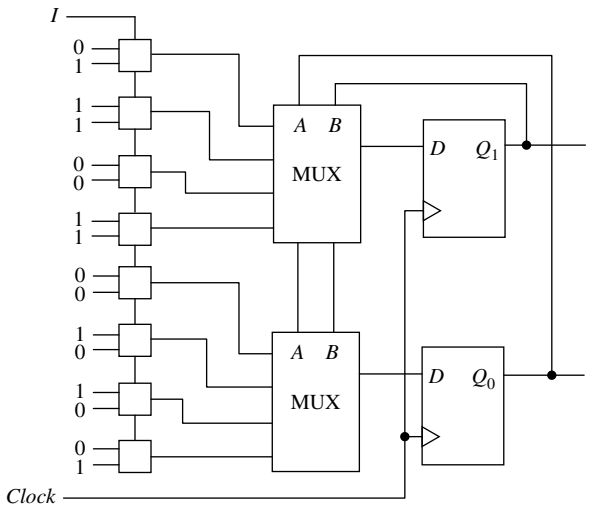


Figure 5.13 Canonical implementation of state table.

Example Consider the machine specified by the following state table and flip-flop state assignments:

	I = 0	I = 1
S_0	S_0	S_2
S_1	S_3	S_2
S_2	S_1	S_0
S_3	S_2	S_3

	Q_1	Q_1
S_2	0	0
S_1	0	1
S_2	1	0
S_3	1	1

This machine can be implemented in the canonical form of Figure 5.13. ■■

5.5 EXPERIMENTS WITH SEQUENTIAL MACHINES

Early efforts at testing state machines consisted of experiments aimed at determining the properties or behavior of a state machine from its state table.[12] Such experiments consist of applying sequences of inputs to the machine and observing the output response. The input sequences are derived from analysis of the state table and may or may not also be conditional upon observation of the machine's response to previous inputs. Sequences in which the next input is selected using both the state table and the machine's response to previous inputs are called *adaptive experiments*.

The selection of inputs may be independent of observations at the outputs. Those in which an entire input sequence is constructed from information contained in the state table, without observing machine response to previous inputs, are called *preset experiments*.

A sequence may be constructed for one of several purposes. It may be used to identify the initial or final state of a machine or it may be used to drive the machine into a particular state. Sequences that identify the initial state are called *distinguishing sequences,* those that identify the final state are called *homing sequences*. A sequence that is designed to force a machine into a unique final state independent of the initial state is called a *synchronizing sequence* (the definitions here are taken from Hennie[13]).

The creation of input sequences can be accomplished through the use of trees in which the nodes correspond to sets of states. The number of states in a particular set is termed its *ambiguity.* The root will usually correspond to maximum ambiguity, that is, the set of all states.

Example Consider the state machine whose transitions are described by the state table of Figure 5.14. Can the initial state of this machine be determined by means of a preset experiment?

The object is to find an input sequence that can uniquely identify the initial state when we start with total ambiguity and can do no more than apply a precomputed set of stimuli and observe output response. From the state table we notice that if we apply a 0, states *A* and *D* both respond with a 1 and both go to state *A*. Clearly, if an input sequence starts with a 0, it will never be possible to determine from the response whether the machine started in state *A* or *D*. If the sequence begins with a 1, a 0 response indicates a next state of *B* or *E* and a 1 response indicates a next state of *A*, *B*, or *C*. Therefore, a logic 1 partitions the set of states into two subsets that can be distinguished by observing the output response of the machine.

Applying a second 1 further refines our knowledge because state *B* produces a 1 and state *E* produces a 0. Hence an input sequence of (1,1) enables us, by working backwards, to determine the initial state if the output response begins with a 0. The 0 response indicates that the initial state was a *C* or *E*. If a second 0 follows, then the machine must have been in state *E* after the first input, which indicates that it must originally have been in state *C*. If the second response is a 1, then the machine is in

	I	
	0	1
A	*A*/1	*C*/1
B	*C*/0	*A*/1
C	*D*/0	E/0
D	*A*/1	*B*/1
E	*B*/0	*B*/0

Figure 5.14 State table.

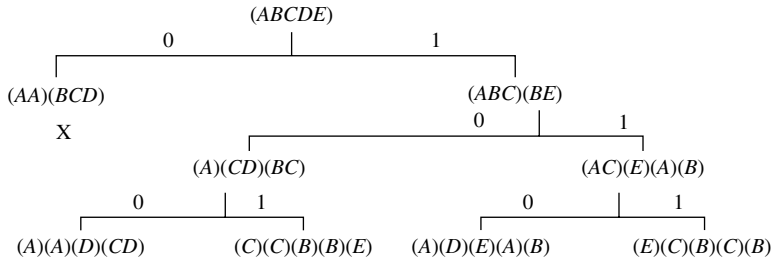


Figure 5.15 Preset experiment.

state B, indicating that it was originally in state E. But what if the initial response was 1? Rather than repeat this analysis, we resort to the use of a tree, as illustrated in Figure 5.15, in which we start with maximum ambiguity at the root and form branches corresponding to the inputs $I = 0$ and $I = 1$. We create subsets comprised of the next states with set membership based on whether the output corresponding to that state is a 1 or 0.

When a 0 is applied to the set with maximum ambiguity, the path is immediately terminated because states A and D merged; that is, they produced the same output and went to the same next state, hence there was no reason to continue the path. When a 1 is applied, two subsets are obtained with no state mergers in either subset. From this branch of the tree, if the second input is a 1, then a third input of either a 0 or 1 leads to a leaf on the tree in which all sets are singletons. If the second input is a 0, then following that with a 1 leads to a leaf in which all sets are singletons. We conclude, therefore, that there are three preset distinguishing sequences of length three, namely, (1, 1, 0), (1, 1, 1), and (1, 0, 1). If the sequence (1, 1, 0) is applied to the machine in each of the five starting states, we get

Start State	Output Response	Final State
A	1 0 0	B
B	1 1 0	D
C	0 0 0	C
D	1 1 1	A
E	0 1 1	A

■■

From the output response the start state can be uniquely identified. It must be noted that a state machine need not have a distinguishing sequence. In the example just cited, if a 1 is applied while in state E and the machine responds with a 1, then another merger would result and hence no distinguishing sequence exists. Another terminating rule, although it did not happen in this example, is as follows: Any leaf that is identical to a previously occurring leaf is terminated. There is obviously no new information to be gained by continuing along that path.

Because the distinguishing sequence identifies the initial state, it also uniquely identifies the final state; hence the distinguishing sequence is a homing sequence. However, the homing sequence is not necessarily a distinguishing sequence. Consider again the machine defined by the state table in Figure 5.14. We wish to find one or more input sequences that can uniquely identify the final state while observing only the output symbols. Therefore, we start again at the source node and apply a 0 or 1. However, the path resulting from initial application of a 0 is not discarded because we are now interested in the final state rather than initial state; therefore state mergers do not cause loss of needed information.

Example We use the same state machine, but only pursue the branch that was previously deleted, since the paths previously obtained are known to be homing sequences. This yields the tree in Figure 5.16.

From this continuation of the original tree we get several additional sequences of outputs that contain enough information to determine the final state. However, because of the mergers these sequences cannot identify the initial state and therefore cannot be classified as distinguishing sequences. ■■

The synchronizing sequence forces the machine into a known final state independent of the start state. We again use the state machine of Figure 5.14 to illustrate the computation of the synchronizing sequence. As before, we start with the tree in which the root is the set with total ambiguity. The computations are illustrated in Figure 5.17.

Starting with the total ambiguity set, we apply 0 and 1 and look at the set of a 1 possible resulting states. With a 0 the set of successor states is (*ABCD*), and with a 1 the set of successor states is (*ABCE*). We then consider the set of all possible successor states that can result from these successor states. From the set of successor states (*ABCD*) and an input of 0 the set of successor states is the set (*ACD*). We continue until we either arrive at a singleton state or all leaves of the tree are terminated. A leaf will be terminated if it matches a previously occurring subset of states or if it properly contains another leaf that was previously terminated. In the example just given, we arrive at the state *A* upon application of the sequence (0, 0, 0, 0). Other sequences exist; we leave it to the reader to find them.

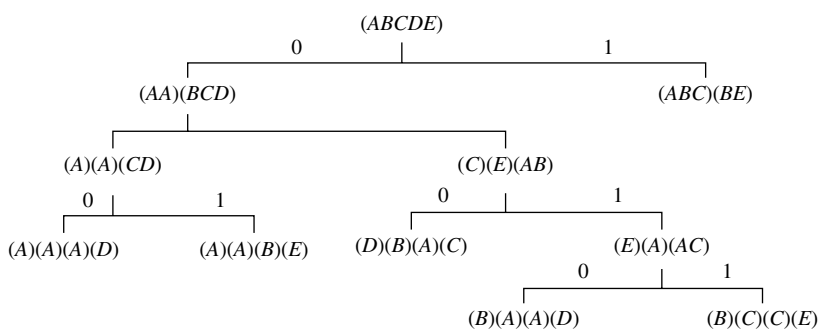


Figure 5.16 Determining final state.

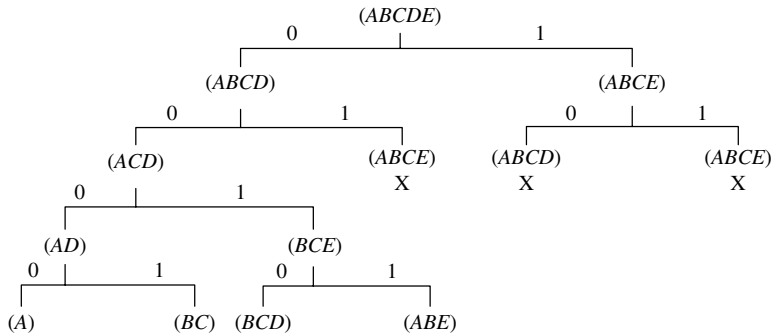


Figure 5.17 Synchronizing sequence.

The same state machine will now be used to describe how to create an adaptive homing sequence. Recall that adaptive experiments make use of whatever information can be deduced from observation of output response. From the state table it is known that if a 0 is applied and the machine responds with a 1, then it is in state *A* and we can stop. If it responds with a 0, then it must be in *B*, *C*, or *D*. Either a 0 or 1 can be chosen as the second input. If a 0 is chosen, we find that with an output response of 1 the machine must again be in state *A* and with a response of 0 it must be in state *C* or *D*. Finally, with a third input there is enough information to uniquely identify the state of the machine. Adaptive experiments frequently permit faster convergence to a solution by virtue of their ability to use the additional information provided by the output response.

The distinguishing sequence permits identification of initial state by observation of output response. This is possible because the machine responds uniquely to the distinguishing sequence from each starting state. The existence of a distinguishing sequence can therefore permit a relatively straightforward construction of a *checking sequence* for a state machine. The checking sequence is intended to confirm that the state table correctly describes the behavior of the machine. It is required that the machine being evaluated not have more states than the state table that describes its behavior. The checking sequence consists of three parts:

1. Put the machine into a known starting state by means of a homing or synchronizing sequence.
2. Apply a sequence that verifies the response of each state to the distinguishing sequence.
3. Apply a sequence that verifies state transitions not checked in step 2.

The state machine in Figure 5.14 will be used to illustrate this. The machine is first placed in state *A* by applying a synchronizing sequence. For the second step, it is necessary to verify the response of the five states in the state table to the distinguishing sequence since that response will subsequently be used to verify state

transitions. To do so, a sequence is constructed by appending the distinguishing sequence (1, 1, 0) to the synchronizing sequence. If the machine is in state *A*, it responds to the distinguishing sequence with the output response (1, 1, 0). Furthermore, the machine will end up in state *B*. From there, state *B* can be verified by again applying the distinguishing sequence.

This time the output response will be (1, 1, 0) and the machine will reach state *D*. A third repetition verifies state *D* and leaves the machine in state *A,* which has already been verified. Therefore, from state *A* a 1 is applied to put the machine into state *C* where the distinguishing sequence is again applied to verify state *C*. Since the machine ends up in state *C*, a 1 is applied to cause a transition to state *E*. Then the distinguishing sequence is applied one more time to verify *E*. At this point the distinguishing sequence has been applied while the machine was in each of the five states. Assuming correct response by the machine to the distinguishing sequence when starting from each of the five states, the input sequence and resulting output sequence at this point are as follows:

s.s	d.s.	d.s.	d.s.	—	d.s.	—	d.s.
input.....0 0 0 0	1 1 0	1 1 0	1 1 0	1	1 1 0	1	1 1 0
output.....	1 0 0	1 1 0	1 1 1	1	0 0 0	0	0 1 1

The synchronizing sequence is denoted by s.s., and the distinguishing sequence is denoted by d.s. The dashes (—) denote points in the sequence where inputs were inserted to effect transitions to states that had not yet been verified. The output values for the synchronizing sequence are unknown; hence they are omitted.

If the machine responds as indicated above, it must have at least five states because the sequence of inputs (1, 1, 0) occurred five times and produced five different output responses. Since we stipulated that it must not have more than five states, we assume that it has the same number of states as the state table. Now it is necessary to verify state transitions. Two transitions in step 2 have already been verified, namely, the transition from *A* to *C* and the transition from *C* to *E*; therefore eight state transitions remain to be verified.

Since the distinguishing sequence applied when in state *E* leaves the machine in state *A*, we start by verifying the transition from *A* to *A* in response to an input of 0. We apply the 0 and follow that with the distinguishing sequence to verify that the machine made a transition back to state *A*. The response to the distinguishing sequence puts the machine in state *B* and so we arbitrarily select the transition from *B* to *C* by applying a 0. Again it is necessary to apply the distinguishing sequence after the 0 to verify that the machine reached state *C* from state *B*. The sequence now appears as follows:

s.s	d.s.	d.s.	d.s.	—	d.s.	—	d.s.	—	d.s.
input.....0 0 0 0	1 1 0	1 1 0	1 1 0	1	1 1 0	1	1 1 0	0	1 1 0
output.....	1 0 0	1 1 0	1 1 1	1	0 0 0	0	0 1 1	0	0 0 0

We continue in this fashion until all state transitions have been confirmed. At this point six transitions have not yet been verified; we leave it as an exercise for the reader to complete the sequence.

5.6 A THEORETICAL LIMIT ON SEQUENTIAL TESTABILITY

The D-algorithm described by Paul Roth[14] is known to be an algorithm in the strictest sense. It can generate tests for combinational circuits, given no more than a structural description of the circuit, including the primitives that make up the circuit and their interconnections. In this section it is shown that such a claim cannot be made for general sequential circuits under the same set of conditions.

The pulse generator of Figure 5.18 demonstrates that this is not true for asynchronous sequential circuits. In normal operation, if it comes up in the 0 state when power is applied, it remains in that state. If it comes up in the 1 state, that value reaches the reset input and resets it to 0 (assuming an active high reset). Since it is known what stable state the circuit assumes shortly after powering up, it can be tested for all testable faults. Simply apply power and check for the 0 state on the output. Then clock it and monitor the output for a positive going pulse that returns to 0.

A simulator that operates on a structural model begins by initializing all the nets in the circuit to the indeterminate X state. The X at the Q output of the self-resetting flip-flop could be a 1 or a 0. If a simulator tries to clock in a 1, both possible states of X at the reset input must be considered. If the X represents a 1, it holds the circuit to a 0. If the X represents 0, it is inactive and the clock pulse drives the output to 1. This ambiguity forces the simulator to leave an X on the Q output. So, despite the fact that the circuit is testable, with only a gate-level description to work with, the simulator cannot drive it out of the unknown state.

For the class of synchronous sequential machines, the Delay flip-flop in which the $\overline{Q}$ output is connected to the Data input, essentially an autonomous machine, is an example of a testable structure that cannot be tested by an ATPG, given only structural information. We know that there should be one transition on the output for every two transitions on the clock input. But, again, when all nets are initially set to the indeterminate state, we preclude any possibility of predicting the behavior of the circuit.

It is possible to define the self-resetting flip-flop as a primitive and specify its behavior as being normally at 0, with a pulse of some specified duration occurring at the output in response to a clock input. That, in fact, is frequently how the circuit is handled. The monostable, or single shot, is available from IC manufacturers as a single package and can be defined as a primitive.

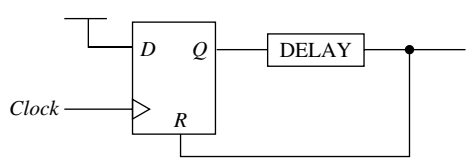


Figure 5.18 Self-resetting flip-flop.

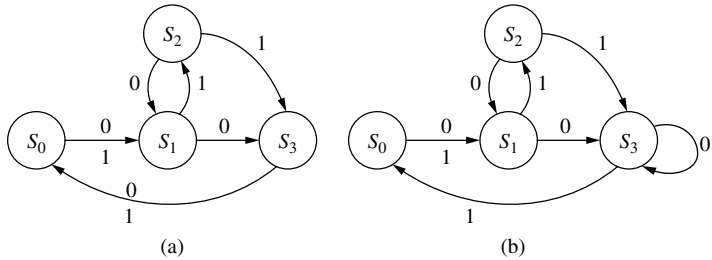


Figure 5.19 State transition graphs.

If the self-resetting flip-flop is modeled as a primitive and if the autonomous machine is excluded, can it be shown that synchronous sequential machines are testable under the same set of conditions defined for the D-algorithm? To address this question, we examine the state transition graphs of Figure 5.19. One of them can be tested by a gate-level ATPG, using only structural information; the other cannot, even though both of them are testable.

The state tables for the machines of Figure 5.19(a) and 5.19(b) are shown in Figures 5.20(a) and 5.20(b), respectively. For machine A the synchronizing sequence $I = (0, 1, 0, 1, 0)$ will put the machine in state S_1. For machine B the synchronizing sequence $I = (0, 0)$ will put the machine in state S_3. The length and nature of the synchronizing sequence plays a key role in determining whether the machine can be tested by a gate-level ATPG. Consider the machine shown in Figure 5.21; it is an implementation of the machine in Figure 5.19(a). Assign an initial value of (X,X) to the flip-flops labeled Q_1, Q_0. Because a synchronizing sequence of length 5 exists, we know that after the application of 5 bits the machine can be forced into state S_1. However, upon application of any single stimulus, whether a 0 or 1, machine A has an ambiguity of at best 3 and possibly 4. Because the ambiguity is greater than 2, two bits are required to represent the complete set of successor states, hence simulation of any binary input value must leave both output bits, Q_1 and Q_0, uncertain; that is, both Q_1 and Q_0 could possibly be in a 0 state or a 1 state, hence, both Q_1 and Q_0 remain in the X state.

Data

	0	1
S_0	S_1	S_1
S_1	S_3	S_2
S_2	S_1	S_3
S_3	S_0	S_0

Data

	0	1
S_0	S_1	S_1
S_1	S_3	S_2
S_2	S_1	S_3
S_3	S_3	S_0

Figure 5.20 State tables.

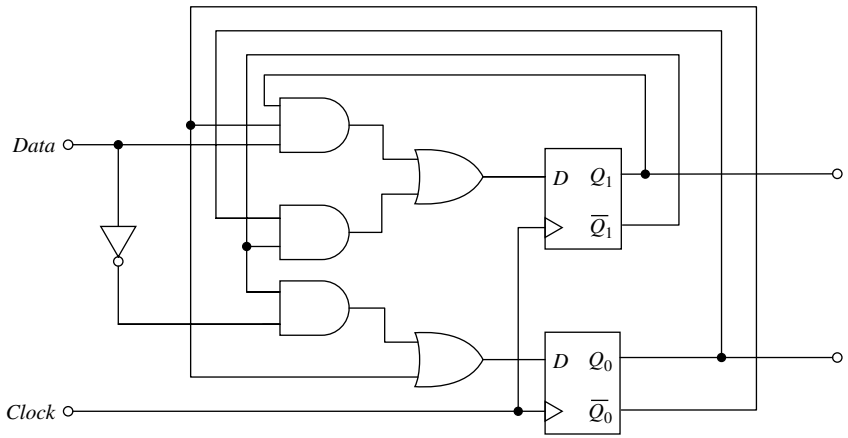


Figure 5.21 Implementation of the state machine.

In general, if a synchronizing sequence exists for an M-state machine, $2^{m-1} < M \leq 2^m$, implemented with m flip-flops, the machine is testable. It is testable because the synchronizing sequence will drive it to a known state from which inputs can be applied that will reveal the presence of structural defects. A synchronizing sequence can be thought of as an extended reset; conversely, a reset can be viewed as a synchronizing sequence of length 1. However, if no single vector exists that can reduce ambiguity to 2^{m-1} or less, then all flip-flops are capable of assuming either binary state. Put another way, no flip-flop is capable of getting out of the indeterminate state.

Given a vector that can reduce ambiguity enough to cause one flip-flop to assume a known value, after some number of additional inputs are applied the ambiguity must again decrease if one or more additional flip-flops are to assume a known state. For an M-state machine implemented with m flip-flops, $2^{m-1} < M \leq 2^m$, the ambiguity must not exceed 2^{m-2}. What is the maximum number of input vectors that can be applied before that level of ambiguity must be attained?

Consider the situation after one input has been applied and exactly one flip-flop is in a known state. Ambiguity is then 2^{m-1}. From this ambiguity set it is possible to make a transition to a state set wherein ambiguity is further reduced, that is, additional flip-flops reach a known value, or the machine may revert back to a state in which all flip-flops are in an unknown state, or the machine may make a transition to another state set in which exactly one flip-flop is in a known state. (In practice, the set of successor states cannot contain more states than its predecessor set.) For a machine with m flip-flops, there are at most $2m$ transitions such that a single flip-flop can remain in a known state, 0 or 1. After $2m$ transitions, it can be concluded that, if the ambiguity is not further resolved, it will not be resolved because the machine will at that time be repeating a state set that it previously visited.

Given that i flip-flops are in a known state, how many state sets exist with ambiguity 2^{m-i}? Or, put another way, how many distinct state sets with i flip-flops in a known state can the machine transition through before ambiguity is further reduced or the machine repeats a previous state set? To compute this number, consider a single selection of i positions from an m-bit binary number. There are $\binom{m}{i}$ ways these i bits can be selected from m positions and 2^i unique values these i positions can assume. Therefore, the number of state sets with ambiguity 2^{m-i}, and thus the number of unique transitions before either repeating a state set or reducing ambiguity, is $2^i \cdot \binom{m}{i}$. Hence, the synchronizing sequence is upper bounded by

$$\sum_{i=1}^{m-1} 2^i \cdot \binom{m}{i} = 3^m - 2^m - 1$$

From the preceding we have the following:

Theorem Let M be a synchronous, sequential M-state machine, $2^{m-1} < M \leq 2^m$, implemented with m binary flip-flops. A necessary condition for M to be testable by a gate-level ATPG using only structural data is that a synchronizing sequence exist having the property that, with i flip-flops in a known state, the sequence reduces the ambiguity to 2^{m-i-1} within $2^i \cdot \binom{m}{i}$ input stimuli.[15]

Corollary The maximum length for a synchronizing sequence that satisfies the theorem is $3^m - 2^m - 1$.

The theorem states that a synchronizing sequence of length $\leq 3^m - 2^m - 1$ permits design of an ATPG-testable state machine. It does not tell us how to accomplish the design. In order to design the machine so that it is ATPG-testable, it is necessary that state assignments be made such that if ambiguity at a given point in the synchronizing sequence is 2^{m-i}, then state assignments must be made such that the 2^i states in each state set with ambiguity equal to i all have the same values on the 2^{m-i} flip-flops with known values.

Example The state machine described in the following table has a synchronizing sequence of length 4. The synchronizing sequence is $I = (0, 1, 1, 0)$.

	0	1
S_0	S_0	S_2
S_1	S_1	S_3
S_2	S_0	S_0
S_3	S_1	S_2

The state sets that result from the synchronizing sequence are

$$\{S_0, S_1\} \rightarrow \{S_2, S_3\} \rightarrow \{S_0, S_2\} \rightarrow \{S_0\}$$

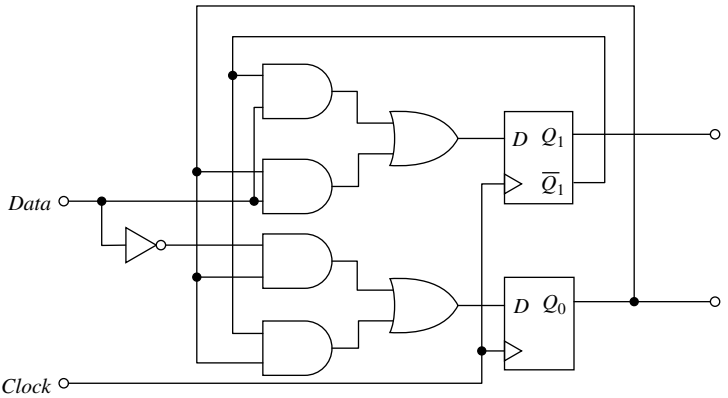


Figure 5.22 Machine with length 4 synchronizing sequence.

If we assign flip-flop $Q_1 = 0$ for states S_0 and S_1, $Q_1 = 1$ for states S_2 and S_3, and $Q_0 = 0$ for states S_0 and S_2, then simulation of the machine, as implemented in Figure 5.22, causes the machine to go into a completely specified state at the end of the synchronizing sequence. ■■

The importance of the proper state assignment is seen from the following assignments.

		Q_1	Q_2
S_0	—	0	1
S_1	—	1	0
S_2	—	1	1
S_3	—	0	0

From the synchronizing sequence we know that the value 0 puts us in either state S_0 or S_1. However, with this set of state assignments, Q_1 may come up as a 0 or 1; the same applies to Q_0. Hence, the synchronizing sequence is not a sufficient condition.

We showed the existence of a state machine with synchronizing sequence that could not be tested by an ATPG when constrained to operate solely on structural information. It remains to show that there are infinitely many such machines. The family in Figure 5.23 has an infinite number of members, each member of which has a synchronizing sequence but, when implemented with binary flip-flops, cannot be driven from the unknown to a known state because the ATPG, starting with all flip-flops at X, cannot get even a single flip-flop into a known state.

	I = 0	I = 1
S_0	S_1	S_0
S_1	S_2	S_0
.	.	.
.	.	.
.	.	.
S_{n-1}	S_n	S_{n-2}
S_n	S_0	S_{n-1}

Figure 5.23 Family of state machines.

5.7 SUMMARY

The presence of memory adds an entirely new dimension to the ATPG problem. A successful test now requires a sequence of inputs, applied in the correct order, to a circuit in which some or all of the storage elements may initially be in an unknown state. New types of faults must be considered. We must now be concerned not only with logic faults, but also with parametric faults, because proper behavior of a sequential circuit depends on storage elements being updated with correct values that arrive at the right time and in the correct order.

Several methods for sequential test pattern generation were examined, including critical path, which was examined in the previous chapter. Seshu's heuristics are primarily of historical interest although the concept of using multiple methods, usually a random method followed by a deterministic approach, continues to be used. The iterative test generator permits application of the D-algorithm to sequential logic. The 9-value ITG can minimize computations for developing a test where a circuit has fanout. Extended backtrace discards the forward trace and aligns sequential requirements by working back from the output, once a topological path has been identified. Sequential path sensitizer extends the D-algorithm to sequential circuits and defines rules for chaining the extended symbols across vector boundaries.

Other methods for sequential test pattern generation exist that were not covered here. In one very early system, called the SALT (Sequential Automated Logic Test)[16] system, latches were modeled at the gate level. Loops were identified and state tables created, where possible, for latches made up of the loops. An extension of Boolean Algebra to sequential logic is another early system not discussed here.[17] More recent sequential ATPG systems have been reported in the literature but have had very little impact on the industry.

Despite numerous attempts to create ATPG programs capable of testing sequential logic, the problem has remained intractable. While some sequential circuits are reasonably simple to test, others are quite difficult and some simply cannot be tested by pure gate-level ATPGs. State machines, counters, and other sequential devices

interacting with complex handshaking protocols make it extremely difficult to unravel the behavior in the proper time sequence. In addition to complexity, another part of the problem is the frequent need for long and costly sequences to drive state machines and counters into a state required to sensitize or propagate faults.

The sequential test problem was also examined from a complexity viewpoint. Synchronizing sequences can be used to show that entire classes of testable sequential circuits exist that cannot be tested within the same set of groundrules specified by the D-algorithm. However, more importantly, designers must understand testability problems and design circuits for which tests can be created with existing tools. In other words, they must design testable circuits. We will have more to say concerning the issue of design-for-testability (DFT) in Chapter 8. Then, in Chapter 12 we will examine behavioral ATPG, which uses models described at higher levels of abstraction.

PROBLEMS

5.1 Using the method described in Section 5.3.2, cut the loops in the D flip-flop circuit of Figure 2.7. Convert it into a pseudo-combinational circuit by creating pseudo-inputs and pseudo-outputs.

5.2 Using the pseudo-combinational DFF from the previous problem, use the ITG and D-algorithm to find tests for the following faults:

Bottom input to gate N1 SA1

Bottom input to gate N4 SA1

Top input to gate N5 SA1

5.3 Attempt to create a test for a SA1 on input 3 of gate 3 of the D flip-flop in Figure 2.7. What is the purpose of that input?

5.4 Find a test for each of the four input SA1 faults on the cross-coupled NAND latch of Figure 2.3. Merge these tests to find the shortest sequence that can detect all four faults.

5.5 Section 4.3.5 defines an intersection table for the values {0, 1, D, $\overline{\text{D}}$, X}. Create an equivalent table for the 9-value ITG. Show all possible intersections of each of the nine values with all the others. Indicate unresolvable conflicts with a dash.

5.6 Redesign the circuit in Figure 5.1 by replacing the DFF with the gated latch of Figure 2.4(b). Cut all loops and use the 9-value ITG to find a test for the fault indicated in Figure 5.1.

5.7 Create a table for the exclusive-OR similar to Tables 5.2 and 5.3.

5.8 Use the critical path method of Section 5.3.4 to find a test for a SA1 fault on the Data input of the D flip-flop in Figure 2.7. Show your work.

5.9 Use EBT to find a test for the indicated fault in the circuit of Figure 5.6. For the state machine, use the circuit in Figure 5.12. Identify the TP, and show your work.

5.10 Substitute a D flip-flop for the JK flip-flop in the circuit of Figure 5.7. Assume the existence of a set input. Duplicate the calculations for the path exercised in the text, using this D flip-flop.

5.11 Show that a SA1 on the top input to B_6 in Figure 5.8 cannot be tested using a combinational ATPG.

5.12 In the circuit of Figure 5.8, replace FF_7 by a primary input. The resulting circuit is now internally balanced. Describe how you would use a combinational ATPG to detect a fault on the bottom input of gate B_2.

5.13 A flip-flop can be made into a scan flip-flop if it has a means whereby it can be serially loaded independent of its normal operation. In such a mode, the output of the circuit acts as an additional input to the circuit, and the input to the flip-flop acts as an additional output (see Chapter 8). The circuit of Figure 5.8 can be made into an internally balanced circuit if one flip-flop is converted to a scan flip-flop. Which one is it? What is the sequential depth of the resulting circuit?

5.14 Using the circuit in Figure 5.24, create state machines for the fault-free and faulty circuits. From the state machines, create a sequence that can detect the SA1 fault.

5.15 Complete the checking sequence for the example that was started in Section 5.5.

5.16 Find a synchronizing sequence for the following state machine:

	0	1
S_0	S_0	S_4
S_1	S_1	S_5
S_2	S_2	S_6
S_3	S_3	S_7
S_4	S_0	S_2
S_5	S_1	S_3
S_6	S_0	S_0
S_7	S_0	S_1

5.17 Describe an algorithm for finding a preset distinguishing sequence.

5.18 The machine (a) below has synchronizing sequence 101. If it starts in state C, and the machine (b) starts in state A, then the input sequence 101 causes

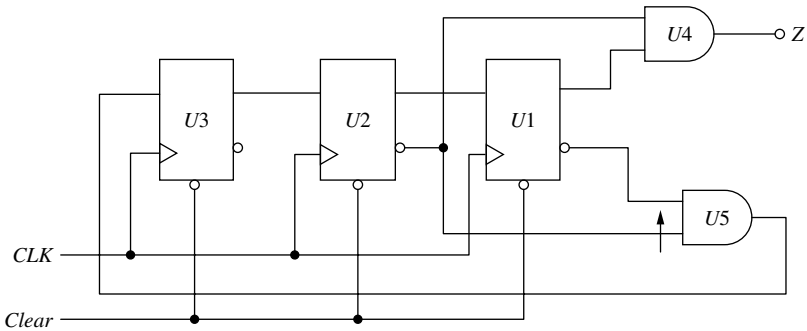


Figure 5.24 Johnson Counter.

identical responses from the two machines. Assuming the application of the sequence 101 to the two machines under the conditions just stated, find a sequence that exercises each state transition in machine (a) at least once, without verification, and causes an identical output response from (b); that is, show that step 2 of the checking sequence is necessary.

	0	1
A	*C*/0	*B*/0
B	*A*/0	*B*/1
C	*B*/1	*C*/1

(a)

	0	1
A	A/0	B/0
B	*C*/1	*C*/0
C	*B*/0	*A*/1

(b)

REFERENCES

1. Seshu, S., On an Improved Diagnosis Program, *IEEE Trans. Electron. Comput.*, Vol. EC-14, No. 2, February 1965, pp. 76–79.
2. Putzolu, G., and J. P. Roth, A Heuristic Algorithm for the Testing of Asynchronous Circuits, *IEEE Trans. Comput.*, Vol. C20, No. 6, June 1971, pp. 639–647.
3. Bouricius, W. G. et al., Algorithms for Detection of Faults in Logic Circuits, *IEEE Trans. Comput.*, Vol. C-20, No. 11, November 1971, pp. 1258–1264.
4. Muth, P., A Nine-Valued Circuit Model for Test Generation, *IEEE Trans. Comput.*, Vol. C-25, No. 6, June 1976, pp. 630–636.
5. Marlett, Ralph, EBT: A Comprehensive Test Generation Technique for Highly Sequential Circuits, *Proc. 15th Des. Autom. Conf.*, June 1978, pp. 332–339.
6. Kriz, T. A., A Path Sensitizing Algorithm for Diagnosis of Binary Sequential Logic, *Proc. 9th Symposium on Switching and Automata Theory*, 1970, pp. 250–259.
7. Kriz, T. A., Machine Identification Concepts of Path Sensitizing Fault Diagnosis, *Proc. 10th Symposium on Switching and Automata Theory*, Waterloo, Canada, October 1969, pp. 174–181.

8. Gupta, R. et al., The BALLAST Methodology for Structured Partial Scan Design, *IEEE Trans. Comput*., Vol. 39, No. 4, April 1990, pp. 538–548.
9. Fujiwara, H. A New Class of Sequential Circuits with Combinational Test Generation Complexity, *IEEE Trans. Comput*., Vol. 49, No. 9, pp. 895–905, September 2000.
10. Balakrishnan, A., and S. T. Chakradhar, Sequential Circuits With Combinational Test Generation Complexity, *Proc. 9th Int. Conf. on VLSI Design*, January 1996, pp. 111–117.
11. Kim, Y. C., V. D. Agrawal, and Kewal K. Saluja, Combinational Test Generation for Various Classes of Acyclic Sequential Circuits, *IEEE Int. Test Conf.*, 2001, pp. 1078–1087.
12. Moore, E. F., Gedanken—Experiments on Sequential Machines, *Automation Studies*, Princeton University Press, Princeton, NJ, 1956, pp. 129–153.
13. Hennie, F. C., *Finite-State Models for Logical Machines*, Wiley, New York, 1968.
14. Roth, J. P., Diagnosis of Automata Failures: A Calculus and a Method, *IBM J. Res. Dev.*, Vol. 10, No. 4, July 1966, pp. 278–291.
15. Miczo, A. The Sequential ATPG: A Theoretical Limit, *Proc. IEEE Int. Test Conf.*, 1983, pp. 143–147.
16. Case, P. W. et al., Design Automation in IBM, *IBM J. Res. Dev.*, Vol. 25, No. 5, September 1981, pp. 631–646.
17. Hsiao, M. Y., and Dennis K. Chia, Boolean Difference for Fault Detection in Asynchronous Sequential Machines, *IEEE Trans. Comput*., Vol. C-20, November 1971, pp. 1356–1361.

CHAPTER 6

Automatic Test Equipment

6.1 INTRODUCTION

Digital circuits have always been designed to operate beyond the point where they could be reliably manufactured on a consistent basis. It is a simple matter of economics: By pushing the state of the art—that is, aggressively shrinking feature sizes, then testing them and discarding those that are defective—it is possible to obtain greater numbers of ICs from a single wafer than if they are manufactured with more conservative feature sizes (cf. Section 1.8 for more discussion on this practice).

This strategy depends on having access to complex, and sometimes very expensive, test equipment. This strategy also depends on being able to amortize tester cost over many hundreds of thousands, or millions, of ICs. As ICs become more complex, running at faster clock speeds, with greater numbers of I/O pins, requirements on the tester become greater. More pins must be driven and monitored. Tolerances grow increasingly tighter, and there is less margin for error. Clock skew and jitter must be controlled more tightly, and the increasing amount of logic, running at ever higher clock speeds, requires the ability to switch greater amounts of current in less time.

Early testers were quite simple: Input pins were driven by stimuli stored in memory. After some predetermined clock cycle the output pins were strobed and their responses compared to expected responses (cf. Figure 6.1). Many early testers were designed and manufactured by end users, particularly mainframe vendors. With time, however, and the increasing complexity of the ICs and PCBs being tested, it became prohibitively expensive to design and build these testers. Companies were formed for the explicit purpose of designing and building complex testers and, although these testers were quite expensive, it was nevertheless more economical to buy than to build in-house.

Over the years, many tester architectures and test strategies have evolved in order to locate defects in ICs and PCBs and provide the highest possible quality of delivered goods at the lowest possible price.This chapter provides a very brief overview of some of the more important highlights and concepts involved in applying test stimuli to digital circuits and monitoring their response. Space does not permit a

Digital Logic Testing and Simulation, *Second Edition*, by Alexander Miczo
ISBN 0-471-43995-9 Copyright © 2003 John Wiley & Sons, Inc.

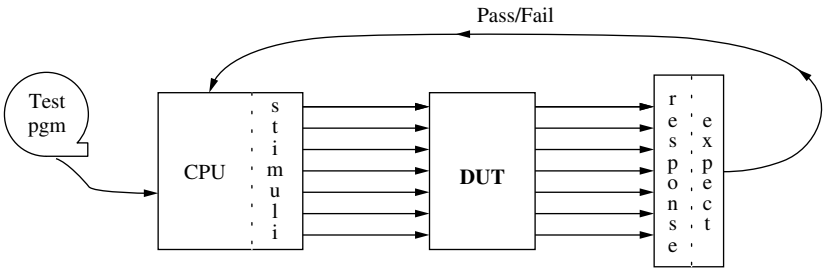


Figure 6.1 Basic test configuration.

more thorough investigation of the many tester architectures and strategies that have been devised to test digital devices during design debug and manufacturing test.

6.2 BASIC TESTER ARCHITECTURES

Functional testers apply stimuli to input pins of a device-under-test (DUT) and sample the response at output pins after sufficient time has elapsed to permit signals to propagate and settle out. The tester then compares sampled response to expected response in order to determine whether the DUT responded correctly to applied stimuli. Depending on their capabilities, these testers can be used to test for correct function, characterize and debug initial parts, and perform speed binning.

6.2.1 The Static Tester

Functional testers can be characterized as static or dynamic. A *static tester*, such as the one depicted in Figure 6.1, applies all signals simultaneously and samples all output pins at the end of the clock period. Device response is compared to the expected response and, if they do not match, the controlling computer is given relevant information such as the vector number and the pin or pins at which the mismatch was detected. The static tester does not attempt to accurately measure *when* events occur. Therefore, if a signal responds correctly but has excessive propagation delay along one or more signal paths, that fact may not be detected by the static tester. These testers are primarily used for go–nogo production testing.

A general-purpose tester must have enough pins to drive the inputs and to monitor the outputs of the DUT. In fact, in order to be general purpose, the tester must have enough pins to drive and sample the I/Os of the largest DUT that might be tested by that tester. Furthermore, since it is not known how many of the I/Os on the DUT are inputs, and how many are outputs, it must be possible to configure each of the tester pins as an input or as an output. If a device has more pins than the tester, it may be possible to extend the capabilities of the tester through the use of clever techniques such as driving two or more inputs from a single tester channel and/or multiplexing IC output pins to a single tester channel where they may be sampled in sequence.

When considering a tester for purchase, its maximum operating speed may be an important consideration, depending on the purpose for which it is being purchased. But other factors, including accuracy, resolution, and sensitivity, must be given equal weight.[1] *Accuracy* is a measure of the amount of uncertainty in a measurement. For example, if a voltmeter is rated at an accuracy of ±0.1% and measures 5.0 V, the true voltage may lie anywhere between 4.95 V and 5.05 V. *Resolution* refers to the degree to which a change can be observed. Referring again to the voltmeter, if it is a digital voltmeter, its resolution is expressed as a number of bits. However, the last few bits may not be meaningful if measurements are being taken in a noisy environment. If the noise is random and there is a need for greater resolution, samples can be averaged. This is done at the expense of sampling rate.

Sensitivity describes the smallest absolute amount of change that can be detected by a measurement. For the voltmeter, sensitivity might be expressed in millivolts or microvolts. Note that these three factors do not necessarily depend on one another. A device may have high resolution or high sensitivity but may not necessarily meet accuracy requirements for a particular application. Moreover, a device may have high sensitivity, but its ability to measure small signal changes may be limited by other devices in the test setup such as the cables used to make the measurements.

Tester programming is another important consideration. Test programs that are used to control testers are normally created on general-purpose computers. They may be derived from design verification vectors, from an ATPG, or from vectors specifically written to exercise all or part of a design in order to uncover manufacturing defects. When the developer is satisfied that the test program is adequate, it is ported to the tester.

The tester will have facilities similar to those found on a general-purpose computer, including tape drives, a modem and/or network card, and storage facilities such as a hard drive. These facilities allow the tester to read a final test program that exists in ASCII form and compile it into an appropriate form for eventual execution on the tester. Other facilities supported by the computer include the ability to debug tester programs on the tester. This may include features such as printing out failing response from the DUT, altering input values or expect values, masking failing pins and switching mode from stop on first failure to stop after n failures, for some arbitrary n.

When the compiled program is needed, it is retrieved from hard disk. The part of the test program that defines input stimuli and expected response is directed to *pin memory.* Behind each channel on the tester there is a certain amount of pin memory capable of storing the stimuli and response for that particular channel. The goal is to have enough memory behind each tester channel to store an entire test sequence. However, testers may allow pin memory to be reloaded with additional stimuli and response from the hard drive. When refreshing pin memory, each memory load may require an initialization sequence, particularly if the DUT contains dynamic parts. Some parts may also run very hot, and the additional time on the tester, waiting for pin memory to be updated, may introduce reliability problems for the part.

Many of the pins on a typical DUT may be bidirectional pins, acting sometimes as inputs and sometimes as outputs. Therefore, on a general-purpose tester, it must

be possible to dynamically change the function of the pins so that during execution of a test a tester channel may sometimes drive the pin that it is connected to, and sometimes sample that same pin. This and other pieces of information must be provided in the test program developed by the test engineer. Other information that must be provided includes information such as voltage and current limits. A subsequent section will examine a tester language designed to configure tester channels and control the tester.

6.2.2 The Dynamic Tester

It is increasingly common for ICs to be designed to operate in applications where, in order to operate correctly with other ICs mounted on a complex PCB, they must adhere closely to propagation times listed in their data sheets. In such applications, excessive delays can be a serious problem. Isolating problems on a PCB caused by excessive propagation delays is especially difficult when all the ICs have passed functional test and are assumed to be working correctly. It is also possible that correct behavior of an IC involves outputting short-lived pulses that are present only briefly but are nevertheless necessary in order to trigger events in other ICs. These situations, excessive delay and appearance of pulses at output pins, are not handled well by static testers. Other challenges to static testers include application of tests to devices such as dynamic MOS parts that have minimum operating frequencies.

To exercise devices at the clock frequency for which they were designed to operate, to schedule input changes in the correct order, and to detect timing problems and pulses, the *dynamic tester* is employed. It is also sometimes called a *high-speed functional tester* or a *clock rate tester*. It can be programmed to apply input signals and sample outputs at any time in a clock cycle. It is more complex than the static tester since considerably more electronics is required. Whereas many functions in the static tester are controlled by software, in the dynamic tester they must be built into hardware in order to provide resolution in the picosecond range.

The dynamic tester solves some problems, but in doing so it introduces others. Whereas the static tester employs low slew rates (the rate at which the tester changes signal values at the circuit inputs), the dynamic tester must employ high slew rates to avoid introducing timing errors. However, high slew rates increase the risk of overshoot, ringing, and crosstalk.[2] Programming the tester also requires more effort on the part of the test engineer, who must now be concerned not only with the signal values on the circuit being tested but also with the time at which they occur. The task is further complicated by the fact that these timings are also dynamic, being able to change on a vector-by-vector basis, as different functions inside the IC control or influence the signal directions and logic values on the I/O pins.

The architecture of a dynamic tester is illustrated in Figure 6.2.[3] The test pattern source is the same set of patterns that are used by the static tester. However, they are now controlled by timing generators and wave formatters. The test patterns are initially loaded into pin memory and specify the logic value of the stimulus or the expected response. The remaining circuits specify when the stimulus is to be applied or when the response is to be sampled. The system is controlled by a master clock

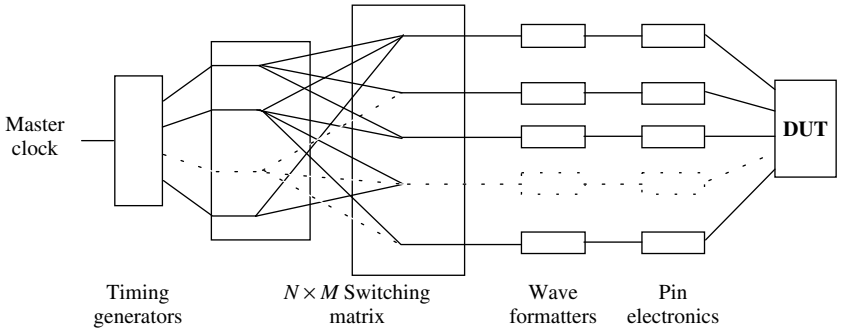


Figure 6.2 Architecture of shared-resource tester.

that determines the overall operating frequency of the board and controls a number of timing generators. Each of the timing generators employs delay elements and other pulse-shaping electronics to generate a waveform with programmable placement of leading and trailing edges. The placement of these edges is determined by the user and can be specified to within a few picoseconds, depending on the accuracy of the tester.

The number of timing generators used in a functional tester depends on whether it is a shared resource or tester-per-pin architecture. A *shared resource* tester (Figure 6.2) contains fewer timing generators than pins and employs a switching matrix to distribute the timing signal to tester pins, whereas the *tester-per-pin* architecture (Figure 6.3) employs a timing generator for each tester pin. Programming the shared resource tester requires finding signals that have common timing and connecting them to the same tester channel so that they can share wave formatters and pin electronics. The switching matrix in the shared resource tester can contribute to skewing problems, so eliminating the switching matrix makes it easier to deskew and thus improve the accuracy of the tester.[4] Another factor that makes the tester-per-pin more accurate is the fact that there is always one fixed-length signal path to the DUT, so the timing can be calibrated for that one path.

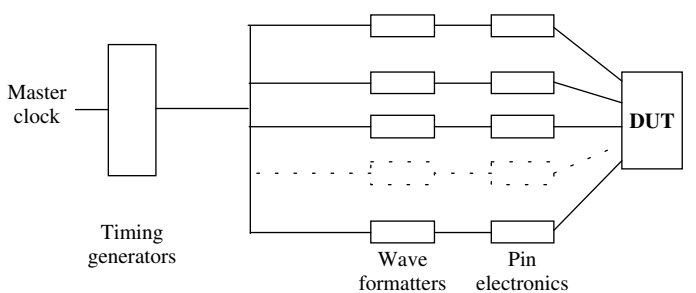


Figure 6.3 Architecture of tester-per-pin tester.

The programming of a tester for a given DUT requires a file containing logic stimulus values to be applied and expected values at the DUT outputs. However, other files are required, including a *pin map* and a file with detailed instructions as to how the waveforms are to be shaped by the pin electronics. The pin map identifies the connectivity between the tester and the DUT. The input stimuli and the expected output responses are stored in tester memory in some particular order. For example, pins 1 through 8 of the DUT may be an eight-bit data path. Furthermore, this data path may be bidirectional. When the pins on the DUT are connected to channels on the tester, it is important that the 8-bit data path on the DUT be associated with the eight channels that are driving or sampling that data path.

6.3 THE STANDARD TEST INTERFACE LANGUAGE

Tester programming languages have tended to be proprietary. Because testers from different companies emphasize different capabilities, it was argued that proprietary languages were needed to fully and effectively take advantage of all of the unique features of a given tester. A major problem with this strategy was that if a semiconductor company owned testers from two or more tester companies, test program portability presented a major problem. If the company wanted to use both of these testers to test a device in a production environment, its engineering staff had to have experts knowledgeable in the test languages provided by each of these testers. For a small company, this could be a major drain on assets, and a single-test engineer might find it difficult to keep up with all the nuances, as well as changes, revisions, and so on, for multiple-test programming languages.

The Standard Test Interface Language (STIL) was designed to provide a common programming language that would let test engineers write a test program once and port it to any tester. It has been approved by the Institute of Electrical and Electronic Engineers (IEEE) as IEEE-P1450.[5] Its goal is to be "tester independent."[6] This is achieved by having the language represent data in terms of its intent rather than in terms of a specific tester.[7] Thus, it is left to the tester companies to leverage to full advantage all of the features of their particular testers, given a test program written in STIL.

STIL provides support for definition of input stimuli and expected response data for test programs. But it also provides mechanisms for defining clocks, timing information, and design-for-test (DFT) capabilities in support of scan-based testing. One of its capabilities is a 'UserKeywords' statement that supports extensibility by allowing the user to add keywords to the language. STIL was initiated as a tool for describing test programs for testers, but its flexibility and potential have made it attractive as a tool for defining input to simulation and ATPG tools. It also offers an opportunity to reduce the number of data bases. Rather than have several data bases to capture and hold data and results from different phases of the design, test, and manufacturing process, STIL offers an opportunity to consolidate these data bases with a potential not only to reduce the proliferation of files, but also to reduce the number of opportunities for errors to creep into the process. Already there is a

growing interest in adding enhancements to facilitate the use of STIL in areas where it was not originally intended to be used.[8]

An example of usage of STIL is presented here to illustrate its use. The circuit will be an 8-bit register with inputs $D_0 - D_7$ and outputs $Q_0 - Q_7$. It will have an asynchronous, active low clear, an active-high output OE, and a clock with active positive edge. When OE is low, the output of the register floats to Z.

Example

```
STIL 0.0;
// 8-bit Reg. with clock and clear
Signals {
  CLK In;
  CLR In;
  OE In;
  D0 In; D1 In; D2 In; D3 In; D4 In; D5 In; D6; In; D7 In;
  Q0 Out; Q1 Out; Q2 Out; Q3 Out; Q4 Out; Q5 Out; Q6 Out;
          Q7 Out;
}
SignalGroups {
  INBUS 'D0 + D1 + D2 + D3 + D4 + D5 + D6 + D7';
  OUTBUS 'Q0 + Q1 + Q2 + Q3 + Q4 + Q5 + Q6 + Q7';
  ALL 'CLK + CLR + OE + INBUS + OUTBUS';
}
Spec timingspec {
  Category prop_time {
    tplh   { Min '2.00ns'; Typ '3.00ns'; Max '4.00ns'; }
    tphl   { Min '2.00ns'; Typ '3.00ns'; Max '4.00ns'; }
    tpzl   { Min '5.25ns'; Typ '6.00ns'; Max '7.00ns'; }
    tpzh   { Min '4.50ns'; Typ '5.50ns'; Max '6.50ns'; }
    tplz   { Min '3.45ns'; Typ '4.20ns'; Max '5.75ns'; }
    tphz   { Min '3.45ns'; Typ '4.20ns'; Max '5.75ns'; }
    strobe_width '3.00ns';
  }

}
Selector typical_mode {
  tplh   Typ;
  tphl   Typ;
  tpzl   Typ;
  tpzh   Typ;
  tplz   Typ;
  tphz   Typ;
```

```
}
Timing timing_info {
  WaveformTable first_group {
    Period '50ns':
    Waveforms {
    CLR { 0 { 'Ons' ForceDown; }}
    CLR { 1 { 'Ons' ForceUp; }}
    OE { 01 { 'Ons' ForceDown/ForceUp; }}
    CLK { 01 { 'Ons' ForceDown/ForceUp;
      CLK_edge: '25ns' ForceUp/Forcedown; }}
    INBUS { 01 { 'Ons' ForceDown/ForceUp; }}
    OUTBUS { L { 'Ons' X; 'CLK_edge+tpzl' l;
          '@+strobe_width' X;}
      H { 'Ons' X; 'CLK_edge+tpzh' h; '@+strobe_width' X;}
      D { 'Ons' X; 'CLK_edge+tplz' t; '@+strobe_width' X;}
      U { 'Ons' X; 'CLK_edge+tpzh' t; '@+strobe_width' X;}
      F { 'Ons' X; 'CLK_edge+tphl' l; '@+strobe_width' X;}
      R { 'Ons' X; 'CLK_edge+tplh' h; '@+strobe_width' X;}
      X { 'Ons' X; } }
    } // end Waveforms
  }  // end WaveformTable first_group
}    // end Timing

PatternBurst stimuli {
  PatList { exercise_part; }
}
PatternExec {
  Timing timing_info;
  Selector typical_mode;
  Category prop_time;
  PatternBurst stimuli;
} // end PatternExec

Pattern exercise_part {
  W first_group;
  // first vector must define states on all signals
  V { ALL=0000000000XXXXXXXX; } // clear the reg's,
                                // don't measure
  V { CLR=1; OUTBUS=XXXXXXXX; }  // release the clear,
                                // don't measure
  V { ALL=0110000000LLLLLLLL; } // outputs enabled
```

```
  V { CLK=0; INBUS=FF; OUTBUS=RRRRRRRR; } // all switching
                                          // to high
  V { INBUS=55; OUTBUS=FHFHFHFH; } // some switch to low
} // end patterns                                        ■■
```

The first line in an STIL program identifies the STIL version. That is followed by a comment. Comments in STIL follow the format employed in the C programming language. A pair of slashes (//) identify a comment that extends to the end of a line. Comments spanning several lines are demarcated by /* ... */.

Immediately following the comment is a block that identifies the I/O signals used in the design. Each signal in the design is identified as an In, Out, or InOut. Signals may be grouped for convenience, using the SignalGroups block. The inputs D0 through D7 to the individual flip-flops of the 8-bit register are grouped and assigned the name INBUS. In similar fashion the outputs of the 8-bit register are grouped and given the name OUTBUS. Then, the entire set of input and output signals are grouped and assigned the name ALL. These groupings prove convenient later when defining vectors.

The Spec block defines specification variables. The Spec block is assigned a name, but it is for convenience only; the name is not used in any subsequent reference. In this example a Category is defined and assigned the name prop_time. Several categories can be defined and used at different places in the test program. Six of the variables in category prop_time are propagation delays that will be used later when defining the WaveformTable. The names of the Spec entries are arbitrary and, in fact, any number of entries could be used in the Spec block. For example, a user may have a legitimate reason to define unique propagation times from X to Z, 0, and 1.

Three values, a minimum, typical, and maximum, are assigned to each of the six variables in the Spec block. A seventh variable called strobe_width has one value that defines the duration of a strobe measurement on an output. The Selector block determines which of the Spec values to use. There are four possibilities: Min, Typ, Max, or Meas. Meas values are determined and assigned during test execution time; they are not explicitly specified in the Spec information.

The Timing block follows the Selector block. It is given the name timing_info. It contains definitions for one or more WaveformTables. In the example presented here there is just one WaveformTable, and it is assigned the name first_group. The first statement assigns a period of 50 ns to all the test vectors that use first_group. Then, some Waveforms are defined. The first one is for CLR, the clear signal. The number 0 follows the signal name CLR. It is called a WaveformChar, abbreviated WFC. Although any character may be used to represent the waveform following the WFC, it is good practice to use a character that has some recognizable meaning because the WFC will be used in the ensuing vectors.

A signal may have several waveforms, but each one must have a different WFC. In STIL a waveform is a series of time/event pairs. In the waveform for CLR the keyword ForceDown follows the time 0 ns. So, at time 0 a ForceDown event occurs; CLR is driven low if it had previously been at a high value. If a signal is in the off (Z) state, it is turned on and driven low. Notice that in the example given above,

there are two waveforms for CLR that have identical timing, so they could actually be merged. However, they were kept separate for illustrative purposes.

Merging is illustrated by the waveform for the output enable OE. At 0 ns OE could switch to either 0 or to 1. Therefore a single WFC 01 represents this time/event pair, and both possibilities are described on that one line. The first entry, ForceDown, corresponds to WFC 0. The second entry, following the slash, corresponds to WFC 1. The character string 01 is called a WFC_LIST.

The next waveform defines the behavior for CLK. Like OE, the CLK signal uses a WFC_LIST. One new thing to note here is the introduction of an event_label definition called CLK_edge. Labels defined in this way are scoped to the WaveformTable in which they are defined. The label is useful in relating subsequent events to the clock edge. The CLK waveform is followed by a waveform for INBUS. It also has a rather simple waveform. However, one distinction here lies in the fact that the waveform applies to all the signals D0 through D7.

The last entry in the WaveformTable is for OUTBUS. Recall that it is the set of outputs Q0 through Q7. There are seven entries for OUTBUS, and each has its own WFC. The first entry for OUTBUS has an L as its WFC. At time 0 ns the tester is told to look for an X on the output. This is simply a way to tell the tester not to measure at this time. Then, at time CLK_edge + tpzl the tester is told to expect *l* (the letter *l*), which is a compare logic low window. In the CLK waveform CLK_edge was defined to occur at 25 ns. So, the tester should start monitoring the OUTBUS at 25 ns + tpzl. Since Typ values were selected by the Selector, and the Typ value for tpzl was defined to be 6.00 ns, the tester should start monitoring at 31.00 ns. The next field begins with the @ symbol. The @ symbol is used to refer to present time, which was defined to be CLK_edge + tpzl in the previous field. So @+strobe_width is 31.00 ns + 3.00 ns, meaning that the tester should continue to monitor OUTBUS until 34.00 ns.

Each of the first six entries for OUTBUS corresponds to one of the six entries in the Spec block. The seventh entry is for those vectors where the output is unknown, and the tester is instructed not to strobe. The letters *l*, *h*, and *t* are called events and indicate a window strobe. The letter *t* is used when the response is supposed to be high impedance during the entire strobe window. Several other events are defined in P1450.

The PatternBurst block, with the name "stimuli," specifies a list of patterns that are executed in a single execution. The example contains one PatList called "exercise_part." There could be several pattern lists, with the user choosing different sets of patterns for different runs. One of the pattern lists could be a common initialization sequence that several designers or test engineers use to ensure consistency across several test programs. The PatternExec follows the PatternBurst block; it contains the commands that pull together all the information needed to perform a test run. The PatternBurst entry is required, the other three entries are optional. If there are multiple entries for Category, Selector, or Timing, then the entry is required in the PatternExec block to avoid ambiguity. In the example above, these blocks only had single entries, so they could have been omitted. It might, however, be good coding practice to include them as reminders for possible expansion of the test program in the future.

We finally come to the list of patterns that will be applied to the DUT. The set of patterns is given the name exercise_part, the same name that appears in the PatList that is part of the PatternBurst block. The first line following the open parenthesis begins with the letter W, it selects the WaveformTable entry that is to be used. The first_group following the W identifies the entry in the WaveformTable. It is used exclusively in this small example, but in a large, complex circuit there could be several WaveformTable entries. Suppose OUTBUS in the above example were bidirectional. Then there would need to be a WaveformTable entry describing its behavior when OUTBUS is acting as an output, and another to describe its behavior when it is acting as an input.

The next entry in the vector list is a comment. A test program, like many other programs, may take on a life of its own, existing for many years after the original creator has gone on to some other calling. It is a good practice to identify what is supposed to be accomplished in each part of a test program, for your benefit as well as some other individual far in the future, since you are the one who may have to debug it or modify it to test an ECO (engineering change order) at some future date.

The V at the beginning of the next line defines one vector. The first vector assigns values to all the inputs and specifies X's on all the outputs. The tester interprets this to mean that it is not required to measure the output values. The next vector causes the CLR to be released. Since the output has not been enabled, the outputs are floating. However, in this example the tester is told not to measure the outputs. On the third vecor the outputs are enabled and the expected response is listed. Notice that in the WaveformTable the CLK signal is 0 for 25 ns and 1 for 25 ns when the WFC is a 0. Hence, this set of vectors has a period of 50 ns. It also should be mentioned that if a signal is not specified in a vector, it retains its last value, so it was not actually necessary to specify CLK = 0 in the fourth vector.

It is beyond the scope of this text to explore all of the capabilities of STIL. The interested reader can consult the IEEE Standard P1450, which contains, in addition to the formal specification of the STIL language, many illustrative examples. As previously pointed out, the language is intended to be independent of any specific tester architecture. It is possible, of course, that a particular program written in STIL calls for capabilities beyond that which a particular tester is capable of, but so long as a tester has the capabilities called for in a particular test program, then it is the responsibility of a compiler provided by that tester vendor to translate the STIL program into a binary form acceptable to the target tester. If an IC manufacturer has several different testers, then, in theory, at least, the same STIL test program should be able to be ported to any of the testers simply by recompiling it. This gives the IC manufacturer much greater flexibility in allocating resources as products mature and needs change.

6.4 USING THE TESTER

Digital testers are used to functionally test ICs and PCBs in order to determine whether they respond correctly to applied stimuli. But testers can also be used to

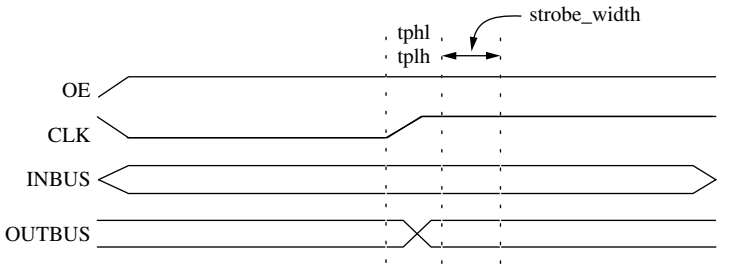


Figure 6.4 Strobe placement.

locate the source of problems, to characterize parts, and to perform speed binning. Consider the example that was used to illustrate the STIL tester programming language. A waveform for the third vector in the example is illustrated in Figure 6.4. The OE signal switches high at the beginning of the waveform, while CLK switches low. Any changes on INBUS also take place at this time. At time 25 ns, CLK begins to switch high. CLK eventually triggers signal changes at the output of the register. The total elapsed time from the beginning of the change on CLK to the time when OUTBUS is strobed is determined by the values in Spec block and Selector block. Although only tphl and tplh are shown in Figure 6.4, there are actually six propagation times listed in the Spec block.

The PatternExec block selected typical_mode from the Selector block. Therefore tplh and tphl values are both 3.00 ns. The strobe_width value, from the Spec block, is given as 3.00 ns. So the tester begins to strobe the OUTBUS at 28.00 ns and continues to strobe until 31.00 ns. OUTBUS is represented here by a single waveform. It could be treated collectively, with all eight signals $Q_0 - Q_7$ strobed at the same time. If a shared resource tester is being used, then all the OUTBUS signals would be driven by the same wave formatter.

If a tester-per-pin tester is being used, strobe placement could be identical for each of the signals $Q_0 - Q_7$, like the shared resource tester, or there could be a unique strobe placement for each signal. With its flexibility, the tester-per-pin might be programmed to strobe all signals concurrently during one vector; then it could be reconfigured on-the-fly to individually strobe the signals on another vector when OUTBUS is being driven by other, unrelated signals. In some proprietary tester programming languages, these programming instructions are called *timing sets* (TSETs).[9]

TSETs can be used to characterize various properties of a device relative to parameters such as voltage, temperature, or clock period. The parameter is varied about some nominal value as a test is applied to the device. An output pin is periodically strobed in order to identify when the pin responds correctly and when it responds incorrectly. A two-dimensional plot called a schmoo is created that characterizes behavior at a particular I/O pin relative to the parameter of interest. This is illustrated in Figure 6.5, where the schmoo shows pass/fail regions at an output pin

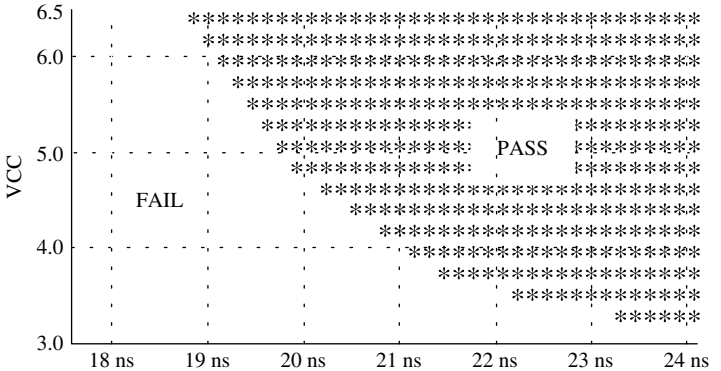


Figure 6.5 A schmoo plot.

as a function of applied voltage. As the voltage decreases, the fail region increases. If the specification for this IC calls for it to function correctly with a 21 ns clock period at 4.0 V, it would just barely meet requirements. Schmoo plots can take on many appearances; for example, the PASS region may be bounded on the right, where the device again fails, yielding an elliptical shape.

When testers apply signals to ICs, they may be programmed to apply logic values specified in pin memory for the entire clock period, or they may be programmed to apply the specified value for part of a period and apply some other value for the remainder of that period. Some commonly used formats include return-to-complement (sometimes called surround-by-complement, or XOR), return-to-zero, return-to-one, return-to-high-impedance, and nonreturn. Figure 6.6 illustrates nonreturn and return-to-one waveforms. Timing generator TG_1 is programmed to go high from 25 ns to 30 ns. Timing generator TG_2 is programmed to go high from 15 ns to 30 ns.

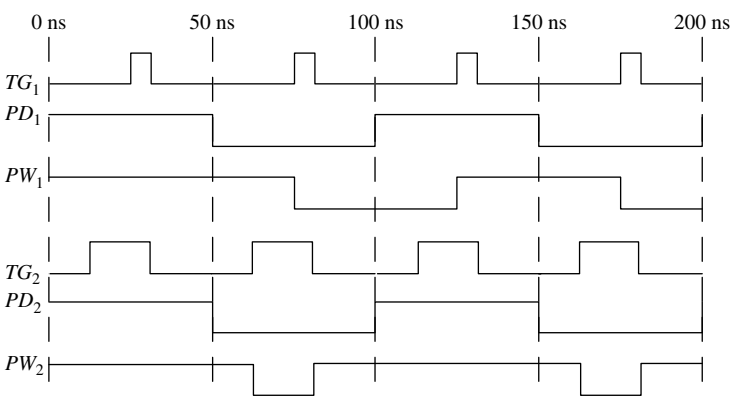


Figure 6.6 Nonreturn and return-to-one waveforms.

Pin data PD_1 and PD_2 are identical; a logic 1 in pin memory is followed by a logic 0, another 1, and then a 0. However, because the timing generators are different and the waveform formats chosen are different, the resulting pin waveforms PW_1 and PW_2 are very different. When PW_1 goes low, it remains low for 50 ns. When PW_2 goes low, it remains low for 22.5 ns. The timing generators determine when the signal changes, but the formatter determines its duration.

As mentioned earlier, complex, high-speed funcional testers are used to test ICs and PCBs to ensure that they operate correctly. But these testers are also being used to characterize new devices. During design, simulators and other electronic design automation (EDA) tools are used at great length to predict how a new design will work, once it is fabricated. However, predicting the behavior of a new technology, always a difficult task, is increasingly complicated by deep submicron effects that were often ignored in earlier technologies.[10] Not only are cell libraries more difficult to characterize, but estimating delay in the wiring between cells must take into account three-dimensional effects that were previously ignored. Guard bands are used to provide a margin of safety during design, to increase the likelihood that the device will operate correctly at its specified clock period. Nevertheless, it is becoming increasingly important to measure critical parameters at speed on a tester to ensure that they respond correctly.

In addition to verifying that a device operates correctly at its specified clock speed, the tester can be used to determine its maximum operating frequency, as well as to generate schmoo plots in order to determine how far the voltage can be dropped before the device fails. Even when the device works correctly at rated speed, the effects of altering clock speed and voltages on noise and crosstalk are difficult to predict with EDA tools.

The engineering test station is targeted to the design engineer. Its design goal is flexibility, in order to allow easy setup of tests, quick change of test parameters, and easy debug. A device can be characterized and debugged on the station, and when the designers are satisfied that the device is working correctly, test information accumulated during this phase is passed on to production, where the priority shifts to maximizing throughput.

One of the parameters that is normally measured on a new device is propagation time. The specification sheet may call for a signal change to occur at an output pin 8 ns after an active clock edge. The output pin may be schmoo'ed in order to determine whether it meets the 8 ns propagation time as well as to determine the margin of error at that pin. After all of the pins are plotted, there is a good database for determining which, if any, pins may represent problems during production.

When characterizing a device on an engineering test station, what happens if the device fails to respond correctly at its intended frequency? The first thing that can be done is to alter the clock frequency. Perhaps the device will operate correctly at a slower frequency. If the device fails to operate correctly at any frequency, then it is logical to assume that there is either a physical failure that occurred during the manufacturing process or a design error. If several parts are available and if all of them fail in an identical fashion, then the logical assumption is that there is a design error that occurred during either the logic design process or the physical design process.

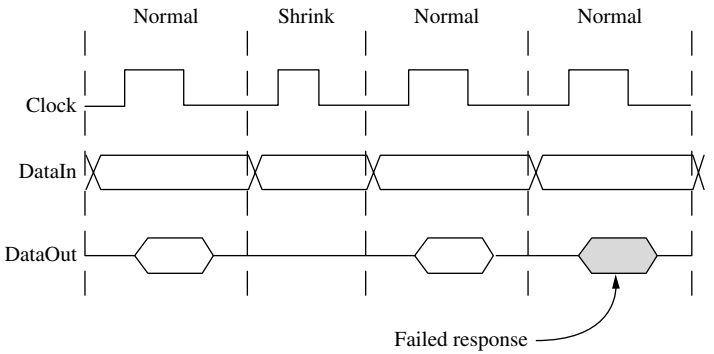


Figure 6.7 Stretch-and-shrink test.

This will require that someone familiar with the logic investigate the response patterns applied by the tester and determine where the defect is most likely to have occurred. At some point it may be necessary to enlist the support of an E-Beam prober to shed more light on the problem (cf. Section 6.5).

But, what happens if the device fails when running at its design frequency, but manages to operate successfully when the clock frequency is lowered? In this case it would be useful to know when the circuit first responds with incorrect results. This can be done by using a *stretch-and-shrink* approach.[11] In this mode of operation, all but one of the test vectors are operated at the slower clock period where the circuit operates correctly. The first time through the vectors, the clock period for the first vector is set to the intended design clock period. If the test passes, then the second vector clock cycle is shrunk and the test is repeated. This is continued until eventually the test program fails. This is illustrated in Figure 6.7, where DataOut is crosshatched. This response may have been induced many vectors earlier by a fault that caused some register or latch to assume an incorrect value.

With a short period on a single preceding vector, and given that the device worked correctly when all the clock periods were applied at normal duration, there is a high likelihood that the incorrect response occurred on the vector with the shrunken cycle. Recall from Chapter 2, where simulation was discussed, that typically only a small percentage of elements in a circuit exhibit logic activity on any given vector. So, knowing on which vector the error occurred can significantly reduce the scope of the search for the problem. In fact, this knowledge, along with information obtained from timing analysis (cf. Chapter 7), can often narrow the search down to just a few critical signal paths. At that point an E-beam can help to further isolate the problem or confirm suspicions as to what path is causing the failure. Armed with this knowledge, the logic designer can approach the redesign effort with greater confidence that the next iteration will be successful.

The stretch-and-shrink test in Figure 6.7 is referred to as the *ripple technique*. Other approaches can also be employed. In the *domino technique*, if the first n test runs are successful, then the clock period for all of those vectors is held at the

shrunken value. It might also be effective to use a variation on a binary search wherein half of the vectors up to the point of failure are run at a shortened clock period in order to expedite the debug process. It is also possible to reverse the entire process, shortening all the clock cycles and then lengthening one or more on each run until the test passes.

The engineering teststation is a powerful tool for characterizing and debugging new designs. It can also be quite useful when it comes time to redesign the product. Existing production units of a device can be evaluated to determine how much margin exists between the specified operating frequency and the target frequency in a redesigned part. The stretch-and-shrink technique can be used to find those vectors where the device begins to fail. That information can be used to help calibrate information obtained from EDA tools. Conservative design rules may have resulted in a device that is being operated far below the maximum frequency at which it is capable of operating.

A successful program for characterizing devices on an engineering workstation requires stimuli that exercise all of the critical paths inside the device, as well as formatting capabilities in order to measure when signals appear at the output pins. These are part of an AC test strategy. But a device that is plugged into a PCB affects its environment. It may place an excessive load on other devices such that they are unable to drive it, or it may have insufficient drive to control other devices. To guard against this possibility, it is necessary to perform DC tests.

The DC test consists of forcing a voltage and measuring current, or forcing current and measuring voltage. This is usually accomplished with the aid of a parametric measurement unit (PMU). It can be mechanically switched to replace a driver or detector that is connected to a pin during normal production test operation. The PMU can force a very precise voltage and measure the resulting current flow, or force a very precise current and measure the resulting voltage. Measurements performed during DC test include power consumption, opens and shorts, input and output leakage, input and output load, and leakage.[12]

When characterizing a device, it is necessary to put the device into a state that permits the desired measurements to be made. A functional program may be run until arriving at a desired output state. Then the measurement is taken. Alternatively, a logic designer or test engineer may write a program whose sole purpose is to drive the circuit into the desired state. For an output leakage test, it is necessary to put the circuit into a state in which the outputs are tri-stated, then measure I_{OZ}, the current at an output when it is in the off-state.

Leakage current I_{IL} is measured by forcing a low-level voltage onto an input by means of the PMU and measuring the current. In similar fashion, leakage current I_{IH} is measured by forcing a high-level voltage onto an input while measuring the current. The high-level output voltage V_{OH} is that voltage which, according to the product specification, corresponds to a high level at the output. V_{OL} corresponds to a low level at the output. V_{OH} is measured by driving the device to a state in which the pin being measured is on, or high, while V_{OL} is measured when the pin is low. Values for these parameters are determined such that the outputs can drive several inputs or loads with adequate noise margin. Guardbands may be established in order to ensure

that the device operates correctly when driving the maximum number of loads in the presence of noise and other environmental factors.

6.5 THE ELECTRON BEAM PROBE

When debugging first silicon, the IC tester can apply stimuli and monitor response in order to determine whether or not the device responds correctly. However, when the response is incorrect, debugging the IC can be a long drawn-out process. This is especially true with respect to a system-on-chip (SOC) that may be comprised of several diverse elements such as CPU, digital signal processor, cache memory, memory management unit, bus control units, and so on. Some of these functional units may have been designed in-house, and some may have been acquired from intellectual property (IP) providers. Some of the acquired units may be soft-core, acquired as RTL code, whereas other units may be hard-core, with only layout and functional specification information provided.

When the device does not work, an error signal may not appear at an I/O pin for many hundreds of clock cycles. When debugging one of these complex devices, it may be impossible to determine the source of an erroneous signal without some visibility into the inner workings of the device, particularly when two or more IP modules are exchanging signals with one another, or even when they are communicating with units designed in-house.

Physical probing of individual die was once possible, when feature sizes were two microns and greater. With shrinking feature sizes and rapidly growing numbers of transistors, physical probing is no longer feasible. With smaller feature sizes the die is more susceptible to damage, and capacitive loading from the probe can distort signals being observed. In addition, the probing process can be extremely time-consuming, tedious, and error prone because the designer must visually distinguish a signal line to be probed from among thousands of such lines that appear nearly identical.

Noncontact probing can be done through the use of the scanning electron microscope (SEM). In this method a die is placed in a vacuum chamber and a focused beam of electrons is directed at the die while the circuits on the die are in operation. The beam is normally blanked (cut off), but is unblanked and allowed to impinge on the die at a time when a voltage sample is desired. When electrons are fired at the die, regions of high voltage attract the electrons while regions of low voltage repel them. A collector captures electrons that are repelled from the surface of the die, and the quantity of electrons captured at a given time is used to estimate the voltage at the point on the surface where the beam was aimed. If the SEM and the device are properly synchronized, the SEM can be used to sample voltages at specified points in several consecutive clock cycles.

Capabilities of the SEM include measurement accuracy of 10 mV with a time resolution of 100 ps.[13] A beam diameter of 0.8 μm can be achieved with a rule of thumb recommending that beam diameter be approximately $W/5$, where W is the width of the interconnections on the die to be investigated.[14] The accelerating

voltage of an e-beam must be limited in order to avoid radiation damage to the device being observed. On the order of 1 or 2 kV is usually suggested as a safe limit.

The method of estimating voltage by collecting electrons repelled from the surface, called voltage contrast, can be used to create waveforms or complete images. In the *waveform* mode the electron beam is pointed at a location on the die and the waveform at that point is constructed by strobing while the die is clocked through a number of states. This mode of operation is quite similar to that of an oscilloscope or logic analyzer. In the *image* mode a picture of the complete die, or some designated part of the die, is constructed by scanning an area of interest. By repeating this operation, several images can be obtained and averaged to minimize the effects of noise and produce a complete image of voltage activity on the top level of the die.

The use of a CAD (computer-aided design) system enhances the efficiency with which e-beam is used. The CAD system may contain physical information describing the die, including the (x, y) coordinates of the endpoints of top-level interconnects. This information can be used to locate particular interconnects on a die and can therefore be used to help position the e-beam accurately. This integration of e-beam, in the waveform mode, together with CAD and a source of input test vectors, then becomes analogous to the printed circuit-board tester. The values on a connector are obtained by the e-beam system and can be compared with expected values derived from simulation to determine if the values on the connector are correct.

The e-beam system is not intended to be used as a production tester. It is slow compared to a conventional tester and may need several hours to acquire enough information to diagnose a problem. The logic states provided by the e-beam at the top-level interconnects may not be sufficient to diagnose problems; analog waveforms at components underneath the top level may also be required. To analyze a die that has already been packaged, it is necessary to de-lid the device, and that is potentially destructive.

The e-beam is best used where short, repetitive cycles of operation can be set up. Nevertheless, it has proven successful for such applications as failure analysis and yield enhancement. When excessive numbers of devices fail with similar symptoms, it is reasonable to expect that the same failure mechanism is causing all or most of the failures. The e-beam may help trace those to design or process errors. If a device operates successfully at some clock frequency but fails when the frequency is increased slightly, it may be possible that a single design factor is limiting performance and that identification and correction of that one factor may permit a significant increase in the clock frequency. The e-beam also proves useful as a research tool to characterize technology and circuit properties.

One of the problems encountered when using e-beam is the fact that it can be difficult to determine which nodes should be probed. If an error is detected at an I/O pin, the fault responsible for the error may have occurred many clock cycles previous to the clock cycle when symptoms were first detected. An approach to solving this problem, called dynamic fault imaging (DFI), uses the image mode to build fault cubes.[15] The fault cube (Figure 6.8) is a series of images from successive machine cycles which are stacked on top of each other to show the origin of a fault and the divergence of error signal(s) in subsequent image frames as a result of that

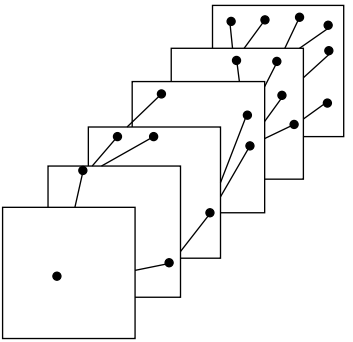


Figure 6.8 Fault cube.

fault. The first step in DFI is to construct voltage contrast images for good and faulty die for several clock cycles. Then the good and faulty device images are differenced to form an image that highlights the areas of the die where different voltage levels exist. On successive clock cycles the fault effects can then be seen to propagate through the die and affect increasing numbers of other states.

The DFI method is under computer control and employs special image processors. It creates a 512×512 image in which each pixel (picture element) is resolved to 8 bits in order to represent a wide range of voltage levels. Pseudocolor lookup tables are used to false color an image so as to enhance visual analysis. As many as 64K images can be averaged to improve resolution. The system has a MOVIE mode in which up to 32 images can be displayed in sequence, either forward or backward in time. A PROBE mode can select the values from the same (x, y) coordinate position of many consecutive images and use these values to construct a waveform corresponding to the voltage at that point on the die. In fact, waveforms corresponding to several (x, y) positions can be created and displayed simultaneously in a logic analyzer format. This kind of integrated design debug system may become routine as more and more complete systems are integrated onto single pieces of silicon.

6.6 MANUFACTURING TEST

To this point the tester has been considered primarily with respect to how it can be used to characterize newly designed devices. However, much of the previous discussion on tester programming and measurement accuracy relates directly to any discussion of manufacturing test. Manufacturing test employs a wide spectrum of instruments in the ongoing effort to distinguish between good and bad products. It uses functional testers, but it also attempts to make use of testers that depend on special probing techniques, including visual inspection. In this section the first step will be to examine the overall test environment. From there we will see how individual test strategies fit into that environment.

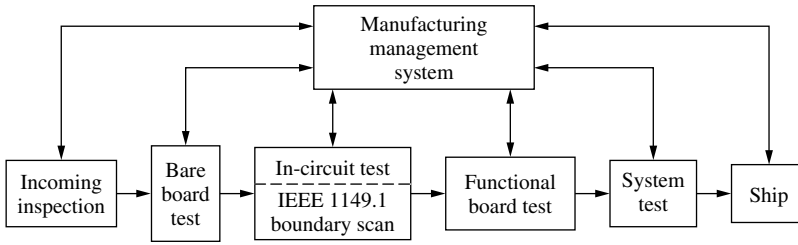


Figure 6.9 The manufacturing test process.

The rule-of-ten guideline introduced in Chapter 1 asserts that the cost impact of a defective component escalates rapidly as it progresses undetected through the manufacturing process. Consequently, the guideline serves as a motivation for detecting defective components as early as possible in the manufacturing cycle.

Manufacturers of complex digital equipment acknowledge the validity of the rule-of-ten by putting in place comprehensive test strategies that distribute test resources throughout the manufacturing process. Testing may begin, as shown in Figure 6.9, with incoming inspection. At this station, components from vendors may be tested to ensure that they comply with some minimum set of specifications. Components may also be exposed to environmental hazards or physical abuse that could induce failures during shipping. A second purpose of incoming inspection is to selectively sort parts. For example, if two or more products use the same IC but one product uses it in a signal path requiring tighter tolerances or faster parts, it may be necessary to sort the parts at incoming inspection and route the parts with preferred characteristics to the design where they are most needed. This is often called speed binning. A thorough screening may, as a beneficial side effect, influence a vendor to improve quality control.

Bare-board testing is employed to detect defects in PCBs before they are populated with components. The object of the test is to verify point-to-point continuity and to check isolation, including high-resistance leakage, between metal runs on the board. Bare-board testers generally use self-learning. In this mode of operation, a tester takes readings between pairs of points on a known good board and stores the results in a file which becomes the test. Multilayer boards may have any number of metal interconnection layers sandwiched between insulating material and connected together by means of through-holes in the insulating material. They can be tested after each metal layer is deposited so that if defects exist, it is still possible to fix them.

The contacts for the measurements are made by means of a *bed-of-nails* fixture. This is a plate in which spring-loaded probes come into physical contact with metal on the PCB. Each of these probes is connected to a driver/receiver pair in the tester so that the probe can either drive a continuity test or monitor the connection between two points. This is illustrated in Figure 6.10 where each trace is contacted by a probe and measurements are enabled. Some manufacturers are starting to use visual recognition

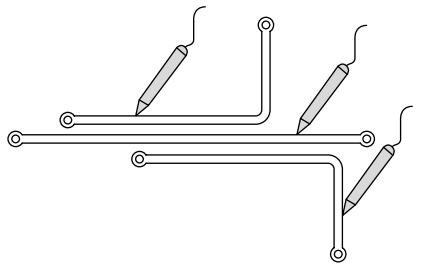


Figure 6.10 Probing traces on a PCB.

systems to detect opens and shorts; however, visual techniques, although capable of higher throughput, cannot quantify resistance and are not as effective at verifying conductivity of through-hole plating.[16]

The boards that pass bare-board test are populated with components. In past years these boards would often be tested with an *in-circuit tester* (ICT). The ICT also uses the bed-of-nails fixture to make contact with electrical points on the board. The board to be tested is placed on a perimeter gasket and then a vacuum is used to pull the board down onto the fixture and into contact with spring-loaded nails or contacts. A wiring harness connects these nails to the tester. When the nails are brought into contact with the board, the tester, under program control, selectively applies signals to some of the nails and monitors others. In this way the tester can test individual components, including ICs, resistors, and inductors used within a circuit.

The ICT is capable of identifying defects introduced during manufacturing. These defects include missing components, wrong components, components inserted with wrong orientation, solder shorts between adjacent pins, and opens resulting from bent pins or cold solder joints. Often several of these defects can be detected in a single pass through the tester. The ICT then prints out a work order explicitly identifying and requesting repair of all the defects. Since the ICT is capable of applying functional tests to integrated circuits, it can also detect failed ICs which, although checked at incoming inspection, might still fail during the manufacturing process from such things as electrostatic discharge or excessive heat.

Note that in Figure 6.9 the ICT shares a box with IEEE 1149.1 boundary scan, often referred to as JTAG (Joint Test Action Group). With packaging techniques making IC connections increasingly inaccessible, it became necessary to find new ways to access connections on the PCB. For this reason the ICT has given way to JTAG on most manufacturing test floors. JTAG will be described in some detail in Section 8.6.2.

From the in-circuit tester, the board goes to a functional tester. This tester applies signals to edge pins and exercises the board as a complete functional entity. Since it is testing the board as a unit, it can detect faults that the in-circuit tester may not detect, including faulty behavior caused by excessive delay. Components may be functionally correct, and individually respond correctly to stimuli, but one or more of them may respond too slowly as a result of parametric faults. The cumulative

delays may alter the order in which two or more signals appear at a device. A slow arriving data or clock at a flip-flop will eventually cause an incorrect value to be clocked in. The dynamic or high-speed functional tester can also detect signals that are too slow in arriving at board edge pins. The functional tester has special facilities for diagnosing fault locations, as well as provisions for margin testing of clock frequency and voltage ranges, features that are useful for detecting intermittents.

After a board has passed board test, either with or without one or more trips to a repair station, it must next be checked out as part of a system. A complete system is assembled and exercised in an operational environment. The problems now encountered include defects resulting from cabling problems, bent pins, high resistance contacts, and erroneous behavior resulting from cumulative delays over two or more boards.

An important component of modern-day manufacturing environments is the manufacturing management system (MMS). The MMS records the manufacturing history of a board during its passage through the production cycle. Information collected on the board includes a history of test results. If a board fails at a particular test station, the cause is diagnosed, it is repaired, and then it is retested. If a board repeatedly fails and is tying up excessive resources, a decision must eventually be made, based on its history, either to continue retesting and repairing it or to scrap it. Information from the MMS can help in making the decision. By compiling statistics on types of defects, and when they occur, the MMS can also help to correct manufacturing processes that are error-prone. In addition, if excessive numbers of boards are incorrectly diagnosed, the MMS may be able to provide an indication that the test for that board must be upgraded.

The MMS can also be used to optimize the overall test strategy. As a product matures, it frequently becomes less prone to manufacturing defects. If statistics indicate that a board rarely fails the in-circuit test, it may become cost effective to bypass the in-circuit test and send the board directly to the functional test station. If, at a later date, the failure rate increases and exceeds some threshold, the MMS can issue a message noting this fact and recommend that boards be routed back through the in-circuit tester.

This strategy may, of course, be modified to execute the in-circuit test and omit the functional test unless a threshold at the functional test station is exceeded. In either case, the optimum strategy must be to use feedback from the MMS to minimize the overall cost of testing. That may mean reducing the amount of capital tied up in expensive test equipment or reducing skill levels required to operate the equipment. The data from the MMS must be periodically reviewed to determine if additional test equipment should be purchased or if it might be more cost effective to move some mature boards away from a particular teststation in order to make it available for new products that must be tested.

6.7 DEVELOPING A BOARD TEST STRATEGY

An effective PCB test strategy is one that finds as many defective devices as possible at the lowest possible cost. The strategy is often flexible, reacting to changing situations

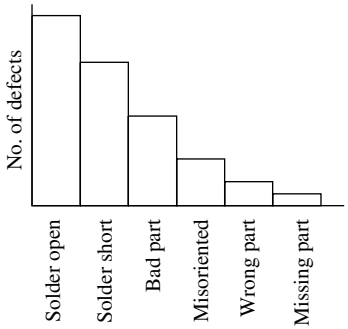


Figure 6.11 Pareto chart.

on the manufacturing floor. Much of that change is dictated by the MMS. IC vendors may be changed due to unavailability of ICs from the original vendor. Processes on the manufacturing floor may be changed to reduce cost. These changes could result in fewer defects, or they could result in more defects. The MMS may spot a link between a new vendor and greater numbers of defects. Alternatively, changing vendors may correct a problem and result in shifting priorities. What was once a major problem becomes a lower priority. Another problem that was once lower priority suddenly becomes the focus of attention. Pareto charts are used to help prioritize problems. The Pareto chart is a bar chart that displays, along the *Y*-axis, a parameter such as number of defects, frequency of occurrence, or total cost of correcting defects. The vertical bars identify different problems relative to the *Y*-axis.

Consider the Pareto chart in Figure 6.11. The first column on the left represents opens that occur during soldering of components onto a PCB. In this Pareto chart it occurs more frequently than any other defect type. Resources addressing this problem will result in a greater number of defect-free PCBs than if some other problem were first addressed. From this chart it might be deduced that solder opens and shorts can possibly be corrected simultaneously. Some judgment is also required because, after analysis, it might be determined that it is a simple, easier matter to fix the problem of missing parts.

The test engineer has at his or her disposal several types of equipment for identifying defective PCBs. Here we consider strategies involving a structural test employing JTAG or ICT plus the functional board tester. In setting up a test floor, the test engineer may be required to choose between a functional board test or a structural test, or the test engineer may adopt both strategies, in which case it is necessary to determine an effective mix of equipment and personnel. The strategy chosen will have a significant impact on manufacturing throughput because boards that reach a system with one or more defects will have to be debugged in the system. A complex system represents significant revenue; if one or more systems must be available at all times to debug faulty boards, then capital is tied up. The object, then, is to minimize the number of faulty boards that reach the system while also minimizing the cost of equipment and labor.

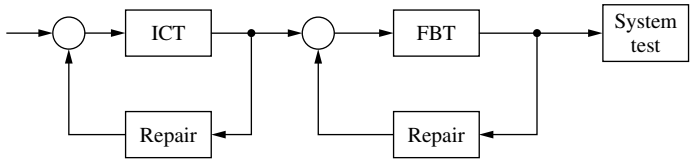


Figure 6.12 Test strategy.

The structural test, as pointed out, is very efficient at finding manufacturing faults; it requires less skill to operate, and test programs are easier to prepare and can be prepared more quickly. In terms of cost of equipment, the JTAG/ICT test is usually cheaper (but an ICT fixture can prove to be a major expense). On the other hand, the functional board tester provides an environment more closely resembling the environment in which the board will ultimately operate. With a good test program, it will find all of the faults that the structural tester will find as well as performance faults that the structural tester will not find. These additional faults are likely to be those that are most difficult to find when a board is plugged into a system.

The types of testing strategies employed are closely related to the volumes of boards manufactured, the number of defects per board, the amount of time required to diagnose and repair defects, and the cost of labor. A common practice is to send boards through the structural tester in order to find the more obvious problems, and then send the boards through the functional board tester (FBT), as illustrated in Figure 6.12.

This strategy uses the structural tester to good advantage to find the most obvious faults at lowest cost; then a functional test is used prior to testing the PCB in a system. If there is high yield at the structural tester, meaning that most faults are found and removed at that station, then most boards will pass at the functional board tester and several structural testers can be used for each functional board tester. If yield from the structural tester is very high, say in excess of 98%, and the system is relatively inexpensive in comparison to a functional board tester, then it may be more economical to omit the functional board test station. Faulty boards that escape detection at the structural tester may be debugged directly in the system. Factored into this approach, of course, must be the cost of more highly trained technicians to debug boards in a system.

Variations on this approach can be employed. If very few PCBs coming from manufacturing are defective, then it may be more economical to test directly at the functional board tester and send failing boards back to the structural tester for diagnosis. After a board has visited the structural tester, if it still fails at the functional board tester, then it might be debugged at the functional board tester.

If it is decided that only one of the two test strategies is to be employed, then the specific objectives of the manufacturing environment must be considered. It is generally accepted that the structural tester can be brought on-line more quickly. If faulty boards coming from the tester are not a problem, either because they can be tested in the system or because they can be discarded if the problem is not quickly

isolated, then the structural tester is probably a good approach. If there are a large quantity of identical boards for which test programs are easily written, or if the PCB must satisfy critical timing requirements, the functional board tester may be the best choice. Regardless of the strategy chosen, the ultimate goal is to limit the number of defective units that reach system test. Diagnosis of faults in complex systems is extremely difficult, hence costly, and there is great economic incentive to limit the number of faulty units that reach system test.

Trade-offs like those discussed for structural test and functional board test also exist when testing components. In this case, though, it is a trade-off between testing die at wafer sort and testing the packaged die. The test at wafer sort is a test of the individual die before they are cut from the wafer. This is often a gross test whose purpose is to identify devices that are clearly dead. The die are marked to indicate whether they passed or failed the test. Those that fail are immediately discarded and those that pass are packaged. Then a more comprehensive package test is performed to ensure that the packaged IC is free of defects.

Wafer sort is directed toward identifying as many defective die as is reasonably possible before incurring the expense of packaging them. There are many die on a wafer, and a 70% yield implies that about a third of them will be defective. In addition, many of those that are defective will fail very early in the test, so it makes sense to apply a brief test that quickly identifies most of those that are defective and discard them before the packaging step is performed. A complete functional test at sort may not identify many more defective die, while subjecting the wafer to a much longer test time.

After the die are cut from the wafer and packaged, a complete functional test can be applied. Even though individual die have been tested while still a part of the wafer, defects can creep in during the packaging process. So, at this stage, before the packaged ICs are shipped to the customer, a complete test of the packaged ICs is performed. Defects that occurred during the assembly process, as well as those faulty die that escaped detection during wafer sort, should be detected here, assuming the fault coverage is adequate.

6.8 THE IN-CIRCUIT TESTER

The third step in Figure 6.9 offers two approaches. The test at this stage may be performed by an in-circuit tester (ICT) or it may be performed by accessing special built-in circuits that support the IEEE 1149.1 standard. For many years the ICT was commonplace on test floors. The dual in-line packages (DIPs) had leads that were physically accessible and the leads were typically 0.10 in. apart. A bed-of-nails fixture came into contact with the PCB, and many manufacturing defects could be diagnosed and repaired quickly in a single pass through the test. This early fault detection can reduce the need for expensive equipment, it can reduce the diagnostic skills required on the part of operators, and it can lower the work-in-process inventory levels.

In recent years, more complex packaging methods have made it virtually impossible to physically access signals on the PCB; as a result, a Joint Test Action Group

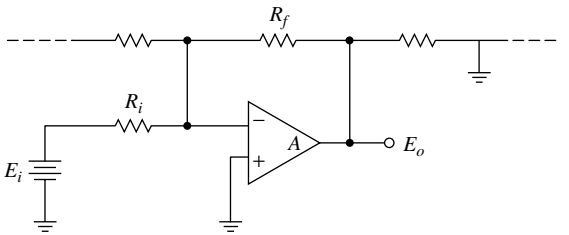


Figure 6.13 The guard circuit.

(JTAG) developed a standard that was eventually accepted by the IEEE (IEEE 1149.1). This will be discussed in Section 8.6.2. A problem with IEEE 1149.1 is the fact that not all ICs support this standard. There is an incentive for PCB manufacturers to support it, but IC manufacturers sometime see it as a cost burden.

The ICT physically probes individual components on the PCB by means of the bed-of-nails and makes use of libraries of tests for individual components. The ICT is able to measure resistances and verify functionality of devices while they are soldered in-place on the PCB. Capacitors can also be tested for shorts. During the test, some devices are backdriven, so tests must be applied for a short duration so as not to damage components while testing other components. When one or more devices is determined to be faulty, a diagnostic message is printed outlining the problem(s) detected, and a work order is issued to repair the board. This approach significantly reduces the cost of initially preparing tests at the board level, as well as the cost of debugging the test, and then, after the test is certified to be correct, the cost of diagnosing and repairing faulty boards.

A functional test can be applied to an IC on the PCB by bringing the bed-of-nails fixture in contact with the board and selectively overdriving individual ICs with large currents while monitoring the IC outputs for correct response. The measurement of resistances makes use of a *guard circuit*.[17] This circuit (see Figure 6.13) employs an op-amp. A known voltage E_i is applied through a precision resistor R_i. The op-amp amplifies the voltage at the (−) terminal and reverses its polarity as it attempts to minimize the voltage difference between its inputs. With a high-gain op-amp the voltage difference is negligible, there is negligible current flow through the op-amp, and the current through R_i is equal to the current through R_f, so the following results are obtained:

$$E_i/R_i = I_x = E_o/R_f$$

Since E_i and R_i are known, R_f can be computed by measuring E_o.

Advantages that have been cited for in-circuit testing include:

Test programming is simplified.

Common manufacturing errors are rapidly detected and diagnosed.

All (or most) faults can be detected in a single pass through the teststation.

Test equipment is cheaper and easier to use.

Test revision due to design changes is usually simpler.

Analog components can be tested.

When forcing voltage levels on IC inputs, the outputs of devices that normally drive the IC are backdriven. This operation can damage the devices as it tests them. Failures can be caused by current densities, and temperature excursions can be immediate or cumulative.[18] The high currents used with in-circuit testers can cause failure in poor wire bonds, but, interestingly, this may be viewed as a desirable side effect since it may precipitate failure of a potentially unreliable bond. Backdriving is a more serious problem when, after a component is tested, it is then backdriven and damaged while testing another component. It is recommended that testing proceed from outputs to inputs in order to test devices after they are stressed. Furthermore, it is recommended that backdriving of low-output impedance devices be avoided.

In-circuit testers are provided with libraries of tests for the more commonly available IC types. However, a test from the manufacturer's library may not be usable because of the manner in which a device is used in a circuit. For example, if an output from a device directly drives one or more of its inputs, that input may become uncontrollable from a test in the library and may necessitate writing a modified test. Clear and set lines, as well as chip select lines, may be tied to power or ground, thus making them uncontrollable.

Precautions may have to be taken even when the test can be applied as it exists on the library. Clock lines on flip-flops and complex LSI devices must be protected from transients which can occur when switching large currents.[19] Buses should receive special attention. All devices driving a bus should first be tri-stated to verify that none of the outputs is faulted in such a way as to pull the bus to a low or high value. Then each device can be tested individually while other devices connected to the bus are inhibited. The inhibit technique can be useful for other devices beside those with tri-state outputs. For example, if the output of a device loops back on itself through a NAND gate, then that feedback can be inhibited by forcing another input of the NAND to a 0.

The in-circuit tester requires a large number of connections from the board under test to the tester; it may require several hundreds or even thousands of wires. The number of wires is held down by assigning a single probe to each net, regardless of how many inputs and outputs are connected to it. At the tester this probe is connected to both a driver and a receiver, which are electronically switched depending on whether the probe is presently driving an input or monitoring an output.

The use of a single probe at each net has an additional advantage in that it increases the probability of detecting an open on a PCB. Consider the net illustrated in Figure 6.14. Suppose that terminals 1 and 2 are connected to tri-state outputs and that terminals 3, 4, and 5 are connected to IC inputs. If a single nail is used and placed in contact with terminal 1, then an open between terminal 1 and 2 will be detected when terminal 2 is monitored and an open will be detected between terminal 1 and any of 3, 4, or 5 whenever any of them is to be driven.

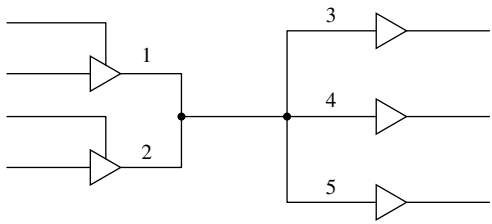


Figure 6.14 Bus with multiple drivers and receivers.

In-circuit testing is not a panacea for all testing problems. It does not detect timing problems. A board may pass the test at an in-circuit station and still fail to perform correctly when plugged into a system. Some devices cannot be backdriven. Others, such as complex VLSI devices, require longer backdrive times, and the duration required may exceed safe limits. Failures that appear at a customer's site are frequently more subtle and less likely to be diagnosed by the in-circuit tester. It is possible that a defective device may cause misleading symptoms; it may pass the in-circuit test but adversely affect another device driving it during actual operation. Shorts between functionally unrelated runs on printed circuit boards may affect operation but go undetected by the in-circuit tester.

The manner in which the circuit board is packaged may prevent it from being tested by the in-circuit tester. A board may contain more nets than the ICT can control. If a board is populated on both sides or if for some other reason nodes are inaccessible, then the in-circuit tester cannot be used. Products that are designed for military use require conformal coating that makes their nodes inaccessible to the in-circuit tester. Some circuits are enclosed within cooling units that make them inaccessible. Dense packaging can make in-circuit test impractical, and some circuits are so sensitive that the capacitance of the in-circuit probe will cause the circuit to malfunction.[20] Future packaging practices, such as (a) complete elimination of boards and (b) three-dimensional wiring, may further restrict the applicability of in-circuit test. For all of these reasons a manufacturing strategy will often require a mix of ICTs and functional testers, as illustrated in Figure 6.12.

6.9 THE PCB TESTER

The growing pervasiveness of digital logic products and their growing complexity, as well as the increasing cost of testing and the need to reduce this cost, has, ironically, sometimes made it necessary to invest more capital in test equipment in order to reduce the overall cost of testing. The objective of improved test equipment is to increase throughput by providing a better test, one that can

Provide high-fault coverage
Run on the tester
Provide good diagnosis

Clearly, a test must provide high-fault coverage. To invest several million dollars in test equipment and highly skilled personnel, and then attempt to distinguish between good and faulty PCBs with a test that has low-fault coverage, can be an exercise in futility, with unacceptable numbers of tester escapes. The ideal goal of a test is to identify specific failed components on a PCB. However, even identifying the existence of a problem, such as a signal path with excessive timing, can save time because it eliminates the need to isolate the problem to a specific board later when testing a complete system.

High-fault coverage, as we have seen in previous chapters, requires good controllability and observability. Controllability may be improved if the functional tester, like the ICT, can backdrive internal points in a circuit. Observability in a PCB can be enhanced through the use of test points. A test may be able to take advantage of socket-mounted ICs that can be removed. With the IC removed, individual pins for that IC become accessible and can be controlled or observed to improve fault coverage and diagnosis.

Printed circuit-board (PCB) testers, like their IC counterparts, are able to create and apply waveforms that are controlled and shaped by pin electronics and formatters. This makes it possible to test PCBs that are functionally the same, but have different timing, using TSETs to compensate for the differences in timing. Complex clock and data patterns can be applied to test not only for incorrect logic response but also for PCBs with excessive delays and missing pulses. However, as we will see, the main feature that distinguishes PCB testers from IC testers is the related hardware that permits the tester to diagnose problems within the board.

6.9.1 Emulating the Tester

High-fault coverage is dependent on the quality of the stimuli, and the ability of the stimuli to take advantage of the controllability and observability of the circuit being tested. However, it is important to note that fault simulation results can be significantly affected by TSETs. A fault simulator can only register detection of a fault if it causes the faulted circuit to differ from the good circuit during the time when an output is being strobed and only if the faulted and good circuits are stable during that period. Therefore, the architecture of the simulator must reflect the architecture of the tester.

This is illustrated in Figure 6.15, where the functional tester is contrasted with the fault simulator. The drive and detect circuitry in the tester use information in the TSETs to schedule primary input changes at the correct time and check for fault detection on primary outputs at times when specific signals are expected. The fault simulator's stimulus or vector file corresponds to the logic 1s and 0s in the tester's pin memory, or drive RAMs.

Just as the tester's detect electronics can be programmed to strobe an output at some specific time, the fault simulator must be able to strobe the output of its circuit model at the same time in order to determine the response of the fault-free circuit as well as to determine if any fault detections occurred. Schmoo plots can be generated during characterization to determine where output signal changes and pulses will

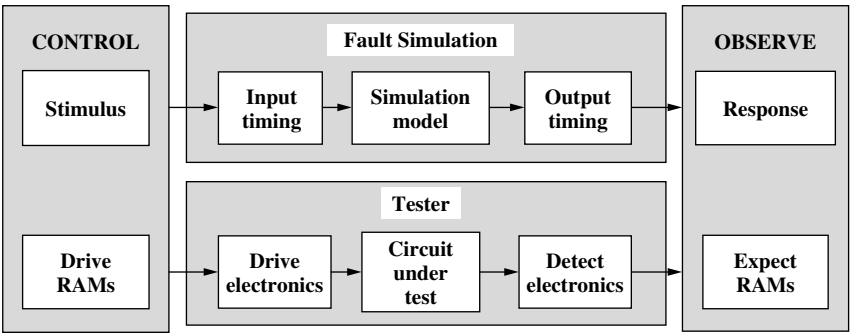


Figure 6.15 Simulation environment versus tester environment.

occur, and both the fault simulator and tester can be programmed to detect not only solid failures but also delay faults.

6.9.2 The Reference Tester

Test stimuli for automatic test equipment can be obtained either from test patterns written by circuit designers and/or diagnostics engineers, or from an ATPG, or from some combination of these sources. The test response can be obtained either by simulating the test stimuli or by running the test stimuli on a reference board and monitoring response. The responses from the reference goard, also called the known good board (KGB) or "golden" board, are recorded in a data file and then used as a standard of comparison for production boards. An alternative approach is to use a tester that can run a test simultaneously on two boards, one of them being the KGB. Then, if there is a miscompare during the test, it is assumed that the production board is faulty.

The KGB approach has the advantage that a test can be written very quickly, with a test for a logic board sometimes being operational within one or two days. However, the approach has some pitfalls, the most obvious being the need to ensure that the KGB is initially free of defects. If running comparison test on two boards simultaneously, the KGB must be maintained in fault-free condition. It may be difficult to hang onto a KGB used for comparison purposes if a complex system, representing a large source of revenue, cannot be shipped to a customer for lack of a circuit board.

When using a KGB, it is necessary to initialize all memory elements on the board to a known value at the start of a test and keep the board in a known state during the test. Random patterns used as test stimuli can create races and hazards, causing unpredictable state transitions, and result in miscompares on boards that are actually good. The failure to initialize a single memory element may go unnoticed for several months if the element is biased to come up in the same state every time. Then, a subtle manufacturing process change, such as rerouting a wire, may change the outcome of a critical race and produce erroneous results several months after a test was thought to be stable.

When using a KGB, it is difficult to provide a qualitative measure of a test, since the estimate of test quality is usually derived from fault simulation. One solution to this problem is to use two KGBs, insert a fault in one of them, and then run the tests to determine if the inserted fault was actually detected. After this is performed for some sufficiently large and representative sample of faults, a fairly accurate measure of fault coverage can be obtained. It is, however, time-consuming and could cause permanent damage to a KGB. Opens are usually harmless to insert, and excessive delays can be emulated with capacitive loading, but inserted shorts could cause a KGB to no longer be a KGB. Furthermore, it is usually not known how the results are affected by engineering change orders. It is also difficult or impossible, when using VLSI components, to emulate many of the faults that occur inside the chip.

6.9.3 Diagnostic Tools

A useful diagnostic tool employed during functional test is the guided probe. It is used when an error is detected at a board edge pin or internal net that is being monitored. Upon detection of an error the guided probe is used to isolate the source of the error. This can be accomplished by either manually or automatically probing selected points on the circuit board. When probing is performed manually, a display device instructs (guides) the operator to contact specified points on the circuit board with a hand-held probe. Automatic probing can be accomplished by means of a bed-of-nails fixture or by a motor-driven probe. The automatic probe requires that the tester have a data file with information on the X, Y coordinates of each pin of each chip on the board relative to a reference point (usually at one corner of the PCB).

The probing operation starts with the board edge pin or internal net at which the tester detects an erroneous signal. From the data base that describes the physical makeup of the board, the tester determines which IC drives the output pin. The tester then

1. Determines which inputs on that IC control the value on the erroneous output.
2. Directs the guided probe to an input of the IC.
3. Runs the entire test while monitoring the values on the input.
4. Repeats steps 2 and 3 for all inputs that affect that output.

If the tester detects an error signal on the output of an IC but does not detect an error signal on any of its inputs, the IC is identified as being potentially at fault. If an erroneous signal is detected on an input at any point during application of the test, then it is assumed that the error occurred at some device between the device presently being probed and the board inputs. Therefore, it is necessary to again back up to the IC that is driving the input pin on the IC currently being checked. This is done until an IC is found with an incorrect output but no incorrect inputs.

The guided probe can be very efficient at locating faulty components. It can help to substantially reduce the skill level required to detect and diagnose most faults on a circuit board because, in theory at least, the operator places the probe on IC pins in response to directions from the tester and then, when the tester detects an IC with a

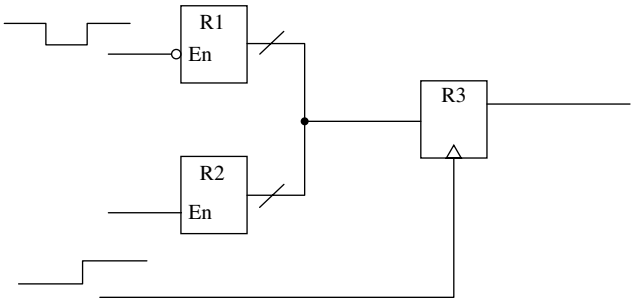


Figure 6.16 Time-dependent data transfer.

wrong output but correct inputs, it instructs the operator to replace that IC. However, it is not foolproof. Consider the circuit of Figure 6.16.

Two tri-state registers are tied together at their outputs and are connected to the inputs of a third register. Register R2 is held in the high-impedance state. Register R1 is enabled for a brief time during the middle of a clock period. While it is enabled, data from R1 is clocked into R3. If erroneous data is found in R3 by the guided probe, it examines the inputs. If it examines the inputs at the end of the clock period when R1 and R2 are both at high impedance, it may conclude that R3 is faulty when, in fact, R3 may have received faulty data from R1.

Notice in the previous paragraphs that a device was declared to be faulty if its output had an error signal but its inputs were correct. In practice, however, it is not quite that simple. If an IC is driving another IC, and the net which interconnects them is SA0 or SA1, it is possible that one of several equivalent faults may have caused the erroneous signal. A fault may exist in the IC which drives the net, or a fault may exist in an IC whose input is connected to the net.

With three or more devices connected to a single net, as in Figure 6.17, resolution of the problem becomes more critical because, if devices are replaced until the board passes the test, a large number of good devices may be unnecessarily replaced before the failing device is discovered. This not only entails several trips to the repair station, but also several passes through the tester, and the entire process of debug and diagnosis may have to be repeated each time. In the meantime, each device removed and replaced increases the possibility of irreparable damage to the board, and there is no assurance that the faulty device will be found.

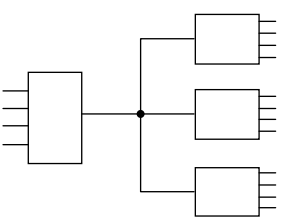


Figure 6.17 Isolating a failing IC.

To help resolve this problem, an electronic knife can be employed.[21] Its purpose is to locate faults internal to a device after the guided probe has identified a net with an erroneous signal. It is capable of employing both DC tests and AC ratio measurements. DC testing measures node resistance by forcing a DC current and measuring the change in DC voltage. If DC tests do not reveal the cause of the problem, then AC ratio measurements are applied. Current is again injected and voltage measurements made at each device connected to the failing net. The device with the lowest impedance is diagnosed as being at fault. This diagnosis assumes that the voltage on a node is controlled by the lowest impedance, and the device controlling the failing net is bad. Success of this measurement technique also rests on the accuracy of the voltage measurements, which in turn depends on the integrity of the test probes, including their physical geometry.

6.10 THE TEST PLAN

A functional board tester requires several files in order to test a circuit board. The data in these files can be classified as test stimuli or diagnostic data. The test stimuli defines the vectors that are applied to every board and can be broken down into data that describe the board test environment and data that define the actual stimuli to be applied. Data that are accessed in response to detection of an error is diagnostic data.

One of the first files generated for a test program is the pin map. This file defines a mapping between I/O pins on the board under test and digital channels on the tester. Its purpose is to ensure that drivers and receivers at the tester drive or monitor the correct signals on the board under test. When test plans are written using symbolic names, these symbolic names will be linked to corresponding channel numbers. It is also necessary to define voltage levels for logic 1 and logic 0, as well as voltage ranges or tolerances, since these values will vary depending on the technology used. In addition, they may vary if it is required that a board be tested at operating margins. A board that normally operates at 5.0 V may be tested at 4.5 V and 5.5 V to determine if it can operate correctly at these voltage extremes. Intermittent errors can sometimes be induced at these marginal voltages.

If debug facilities such as the guided probe and electronic knife are available, then effective use of these resources require that the tester have knowledge of each physically accessible IC pin and test points, including their physical location and the expected logic values for each input vector applied. As with edge pins, the tester may require information defining the probe voltage levels corresponding to logic 1 and 0.

A circuit interconnection file is necessary if a guided probe is used to trace error signals from an output pin back toward board inputs. The interconnection file describes all connections between ICs. A second file that is useful in conjunction with the guided probe is one that lists all inputs that affect each output of each IC on the board. In a circuit such as that depicted in Figure 6.18, the middle input to U1 was supposed to be a 1, but a 0 was detected by the tester. Rather than probe all of the inputs to U2, it is only necessary to probe those inputs that are in the cone of

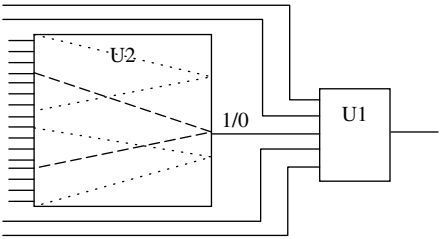


Figure 6.18 Optimizing guided probe operation.

logic that affects U1. This file reduces the number of measurements required and thus cuts down on the number of probe errors. This is particularly important when probing with a hand-held probe, on a densely populated board, since such boards are especially susceptible to misprobes.

Fault dictionaries (cf. Section 7.7.10) were once a popular approach to debugging PCBs. However, the immense amount of data required to diagnose failures in present-day PCBs makes it impractical to employ this approach for any but the smallest circuits. For PCBs that use large ICs, simulation is often impractical. In order to compile a response file for internal nodes, it may be necessary to employ response learning by capturing circuit response at each internally accessible node for the entire duration of the test.

This can be accomplished using the same method that is used to probe the PCB when attempting to diagnose the cause of failures. A probe is brought into contact with each internally accessible node, and the test is run in its entirety. Response is captured and stored at the end of each clock period, to be later used as part of the diagnostic operation. Caution is required here. If simulation is used, unitialized nodes or nodes whose values are indeterminate because of races or hazards can be identified by the simulator. However, capturing response by probing internal nodes during each clock period may result in recording unstable values that differ from one PCB to the next, or from one lot to the next. Good communications between the design team and the test team are important in resolving problems related to initialization.

6.11 VISUAL INSPECTION

Up to this point we have considered testing in the context of applying stimuli and monitoring response. However, many defects can be detected by visual inspection. It was estimated that in 1997 approximately 40,000 people were employed to visually inspect PCBs for errors.[22] Unfortunately, the track record for visual inspection by humans has been rather poor. When two or more people inspect the same PCB under identical conditions, they tend to agree less than half the time. As a result, other inspection methods are being developed to improve on this record.

Automated optical inspection (AOI) has been used effectively. It offers better consistency than humans, who are prone to errors due to fatigue and boredom. AOI captures a visual image of a PCB and stores this in computer memory. Then, production PCBs are scanned and the image is compared to the stored image. While it is not susceptible to errors that humans are prone to, it is nevertheless limited to line-of-sight inspection. It is also susceptible to changes in reflection, possibly caused by boards that are warped or by residues remaining on the PCB, which can cause a high false reject rate.

Infrared thermography is another method being used for visual inspection.[23] Scanning cameras detect invisible infrared radiation emitted by an object or group of objects during test. Electro-optics in the scanner convert this radiation into video signals for display on a monitor. A 256-color palette permits identification of the temperature of the object being scanned. Since failure rates increase exponentially as temperature rises, infrared scanning can detect not only failures, but potential reliability problems at nodes where the circuit responds correctly but may be subject to possible imminent failure due to elevated temperatures.

An advantage of scanning cameras over other means of measuring temperature, such as the use of thermocouples, is their ability to measure temperature without the need for physical contact. Not only does this speed up the measurement process, and make it possible to examine a greater number of nodes, but scanning does not conduct heat away from a junction while the temperature is being read. Temperature accuracy for the infrared thermography cameras is reported to be within ±2°C. In addition to its use for spotting elevated temperatures that may indicate the existence of defects or reliability problems, the data can also be used to suggest redesign in areas of the PCB where everything works as intended, but the circuit runs too hot because of the proximity of devices to one another.

Another technique being used for visual inspection is X rays. One advantage of automated X-ray inspection (AXI) is its ability to see through a PCB and thus inspect both sides of a PCB simultaneously. This has obvious advantages when both sides of a PCB are populated with components. Energy levels of the X rays are chosen so as to be able to pass through materials such as silicon and copper, but be absorbed by solder. Thus, the X rays are able to penetrate such things as RF shields. A major application of AXI is the inspection of solder joints. A ball grid array (BGA) contains many small balls of solder on the underside of the chip. When the chip is placed on the board, the solder is reflowed, causing connections to be made to the PCB. Problems that can occur with the reflow process include missing solder, insufficient solder, improper solder placement, and solder bridges.[24]

The image created by an X ray is a dark round circle where the solder appears. If two solder balls short out during reflow, the solder bridge between the two balls is dark. If an IC is not precisely placed on a PCB, the solder will not line up perfectly with the pads on the PCB. In either of these cases, computer enhancement of the image generated by the X ray will reveal the problems. Solder voids can also be identified. These occur when volatile compounds are trapped inside the solder. During solder reflow the compounds vaporize and pop through the solder, producing the voids. AXI can also detect missing or misaligned components, as well as incorrect orientation of polarized capacitors. Some AXI systems can have difficulty identifying

opens caused by a failure of the solder ball to make contact with the pad. In general, hairline cracks can be difficult for the AXI to detect.

AXI systems usually can only take an image of part of a PCB. A computer can then evaluate the acquired image against a stored image to determine if there are any problems. Then it can automatically reposition the PCB for the next image. Some systems can move the PCB up or down relative to the X-ray source. This causes a change in magnification of the PCB. The PCB can also be rotated so that oblique views of a PCB can be obtained.This permits examination of interior plated-through connections.

Yet another method for detecting faults is time-domain reflectometry (TDR). It can be used to determine where a signal pin is open or shorted and to measure the length of an electrical path.[25] A digital sampling oscilloscope (DSO) equipped with a TDR module is used. The TDR module generates a voltage edge with a fast rise-time, and the DSO records that edge and the signals reflected back to the TDR. These reflections constitute a waveform that can be stored and later recalled during testing to compare with waveforms obtained at suspect nodes on a PCB. Figure 6.19 contains waveforms taken under different circumstances.

The probe tip contact point identifies the time at which the probe tip causes some of the signal to be reflected. However, most of the signal is reflected at the end of the signal path. The distance of the signal path can be measured using half the time from the probe point to the end of signal. Half the total propagation time is 230 ns. Using 1.4×10^8 m/s as the velocity of propagation in copper yields a distance of 16.1 mm from the signal pin to the end of the signal path. A waveform for a failing unit is also illustrated in Figure 6.19. The energy is reflected from an open in the substrate much earlier than expected from the signal in the fault-free circuit. The point at which the reflected signal starts to rise can help to pinpoint the location of the open in the circuit. Waveforms can be obtained from unassembled substrates to further help in isolating opens.

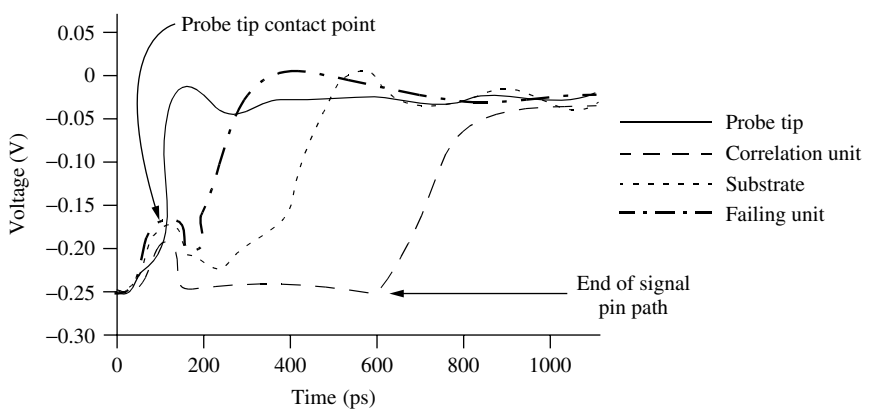


Figure 6.19 TDR comparison of waveforms.

6.12 TEST COST

In coming chapters we will examine methodologies for designing circuits so as to make them easier to test. We end this chapter with some data that give a breakdown on test system cost and follow that with some suggestions for reducing the cost of device test.

In a study published in 1995, Hewlett-Packard looked at all of the factors that contributed to total system test cost. Their cost breakdown findings were as follows:[26]

25% Purchased hardware
12% Purchased software
22% Labor cost of software development
12% Labor cost of hardware development
10% Fixturing
19% Other

A conclusion of the Hewlett-Packard study was that the cost of hardware, however expensive, was only a fraction of total test cost. Suggestions for reducing the cost of device test include the following:[27]

1. Obtain a system with high calibration stability.
2. Include test modes in the circuit under test.
3. Standardize on key suppliers.
4. Use optimum program development tools.
5. Optimize test programs.
6. Upgrade system components (e.g., CPUs), when possible.
7. Use dual test-head systems if possible.
8. As products mature, reduce test program length.

Some of the suggestions are obvious. Others, such as item 2, will be discussed in the following chapters. Some of the items directly touch on cost of ownership. For example, if throughput can be enhanced by means of newer, faster, or more flexible equipment, overall system cost can be amortized over many more devices to be tested, thus reducing test cost per unit. Optimizing test programs may not be so obvious. The goal is to find defective devices as soon as possible. That is where fault simulation comes in. If fault simulation reveals that one test is more effective than another for finding faults, that test should be run first. The goal is always to find defective devices as early as possible in the test cycle. Eventually, as indicated by item 8, the less effective test may eventually not be needed at all as products and processes mature.

6.13 SUMMARY

Tester architectures represent a complex and ever-changing field. It is impossible to do justice to such a diverse topic in a brief chapter. In addition to tester-per-pin

testers, there are also sequencer-per-pin testers, which are more capable, more elaborate, but also more expensive. In addition, the ongoing quest to make ICs smaller and more dense has resulted in increasing numbers of ICs that contain memory and analog functions in addition to digital circuits. Like the digital circuits, these memories and analog circuits must be tested.

Manufacturing processes constantly evolve, putting more circuits on a given die size at the same, or lower, cost. However, this concentration of circuitry exacerbates the test function. The net result is that now the cost of test may consume half or more of the total cost of an IC by the time it is shipped. This cost includes nonrecurring expenses such as the testers and the cost of fixures. Recurring costs include the cost of running the test programs and diagnosing problems. The high skills levels required to run this entire operation imply the need for constant training and upgrading of skills.

Ever-increasing clock speeds of digital devices add another dimension to the test problem. Testers must run faster in order to characterize and test these faster ICs. Speed binning to find the fastest ICs depends on the tester being able to operate at high speeds. These fast testers must be calibrated more often in order to guarantee accuracy at speed and to avoid false negatives—that is, causing good ICs to fail a test and be rejected.

Users attempt to economize on test cost by testing multiple devices in parallel. However, the payback is not linear. Many failures occur on the first few vectors, at which time the test is usually halted and the device discarded. So, for example, when testing two devices individually, one of which is good and the other is bad, the total test time may be 10% greater than the test time for one good device. When testing those two devices in parallel, the test must run to completion. So the savings in test time may be only 10% over the test time when testing the devices individually.

Tester languages have always been a source of confusion. Testers from different companies have traditionally employed unique, proprietary programming languages. STIL may help to alleviate some of the confusion. Only time will tell if it will be embraced by the test equipment community. A previous attempt by the Department of Defense (DOD) to develop a standard test language resulted in ATLAS (Abbreviated Test Language for All Systems).[28] The goal of ATLAS was to define a test in terms of the product to be tested without regard to the tester being used. It, in effect, defines the test for a virtual machine. If a particular tester has a compiler for ATLAS, it can run the test.

The ATLAS language, like STIL, has a preamble that defines the test environment, followed by a procedural section that specifies stimuli and response. It permits testing of digital and analog devices and contains numerous constructs for looping and program control, as well as a specific command to leave the ATLAS language so that the user can use non-ATLAS commands to support capabilities which cannot be supported in the ATLAS language.

PROBLEMS

6.1 Write a STIL program for the test in Figure 6.4 that is used to check for timing compliance (i.e., using tsets to check for critical timing paths).

6.2 In the example of Section 6.3, suppose the 8-bit Register has bidirectional outputs and a selector input that enables it to load the register from the D inputs or from the bidirectional pins when the output is disabled. Modify the STIL program to reflect this capability.

6.3 In the example, Section 6.3, CLR and OE have identical waveforms. Using that observation, how would you rewrite the example to make it more concise?

6.4 In the example, Section 6.3, identify the strobe start and stop times for each of the seven entries for OUTBUS in WaveformTable.

6.5 Describe how you would write a STIL program to implement (1) a stretch-and-shrink test program and (2) a schmoo plot.

6.6 Given a process with 70% yield. Assume that you have a test that covers 100% of the faults, but takes 6 s to run. Also assume that you have 200 die on a wafer. Assume that fault coverage for the test is 66.6%, 82.7%, 89.6%, 94.7%, 98.6%, and 100% after 1, 2, 3, 4, 5, and 6 s, respectively. Finally, assume that the cost of packaging is $.10 per die and that tester time is $.10 per second at both sort test and package test. Determine a strategy to minimize total test cost.

REFERENCES

1. DeSantis, T., Resolution versus Accuracy versus Sensitivity: Cutting Through the Confusion, *Eval. Eng.*, December 1998, pp. 10–16.
2. Sulman, D. L., Clock-Rate Testing of LSI Circuit Boards, *Proc. 1978 IEEE Test Conf.*, pp. 66–70.
3. Catalano, M. et al., Individual Signal Path Calibration for Maximum Timing Accuracy in a High Pincount VLSI Test System, *Proc. Int. Test Conf.*, 1983, pp. 188–192.
4. Bierman, H., VLSI Test Gear Keeps Pace with Chip Advances, *Electronics*, April 19, 1987, pp. 125–128.
5. Standard Test Interface Language (STIL) for Digital Test Vector Data, IEEE-P1450, Draft 0.9, May 1997.
6. Taylor, T., Standard Test Interface Language (STIL): Extending the Standard, *Proc. Int. Test Conf.*, 1998, pp. 962–970.
7. Taylor, T., and G. A. Maston, Standard Test Interface Language (STIL): A New Language for Patterns and Waveforms, *Proc. Int. Test Conf.*, 1996, pp. 565–570.
8. Biggs, N., STIL: The Device-Oriented Database for the Test Development Lifecycle, *Proc. Int. Test Conf.*, 1999, p. 1149.
9. Levin, H. et al., Design of a New Test Generation System for Performance Testing of LSI Digital Printed Circuit Boards, *Proc. Int. Test Conf.*, October 1982, pp. 541–547.
10. Walker, M. G., Modeling the Wiring of Deep Submicron ICs, *IEEE Spectrum*, March 2000, Vol. 37, No. 3, pp. 65–71.
11. Bego, P. M., The Value of an Optimized Engineering Test Station, *Eval. Eng.*, November 1998, pp. 12–25.

12. Stevens, A. K., *Component Testing*, Chapter 4, Addison-Wesley, Reading, MA, 1986.
13. Goto, Y. et al., Electron Beam Prober for LSI Testing with 100 PS Time Resolution, *Proc. Int. Test Conf.*, October 1984, pp. 543–549.
14. Kollensperger, P. et al., Automated Electron Beam Testing of VLSI Circuits, *Proc. Int. Test Conf.*, October 1984, pp. 550–556.
15. May, T. C. et al., Dynamic Fault Imaging of VLSI Random Logic Devices, *Int. Rel. Physics Symp.*, April 1984.
16. Shapiro, D., Universal-Grid Bareboard Testers Offer Users Many Benefits, *Electron. Test*, July 1984, pp. 88–94.
17. Schwedner, F. A., and S. E. Grossman, In-Circuit Testing Pins Down Defects in PC Boards Early, *Electronics*, September 4, 1975, pp. 98–102.
18. Sobotka, L. J., The Effects of Backdriving Digital Integrated Circuits During In-Circuit Testing, *Proc. Int. Test Conf.*, November 1982, pp. 269–286.
19. Mastrocola, Aldo, In-Circuit Test Techniques Applied to Complex Digital Assemblies, *Proc. Int. Test Conf.*, 1981, pp. 124–131.
20. Miklosz, J., ATE: In-Circuit and Functional, *Electron. Eng. Times*, January 3, 1983, pp. 25–29
21. Miczo, A., *Digital Logic Testing and Simulation*, Chapter 6, John Wiley & Sons, New York, 1986.
22. Runyan, S., X-Ray May be PC-Board Key, *Electron. Eng. Times*, April 21, 1997, p. 52.
23. Smith, D., Infrared Thermography Maintains PCB Reliability, *Test Meas. Europe*, Autumn 1993, pp. 33–34.
24. Titus, J., X-Ray Systems Reveal Hidden Defects, *Test Meas. World*, February 1998, pp. 29–36.
25. Odegard, C., and C. Lambert, Reflectometry Techniques Aid IC Failure Analysis, *Test Meas. World*, May 2000, pp. 53–58.
26. Business Trends, Hardware Is Fraction of Total Cost, *Electron. Bus. Today*, December. 1995, p. 26.
27. Iscoff, R., VLSI Testing: The Stakes Get Higher, *Semicond. Int.*, September 1993, pp. 58–62.
28. *IEEE Standard ATLAS Test Language*, IEEE, New York, 1981.

CHAPTER 7

Developing a Test Strategy

7.1 INTRODUCTION

The first five chapters provided a survey of algorithms for logic simulation, fault simulation, and automatic test pattern generation. That was followed by a brief survey of tester architectures and strategies to maximize tester effectiveness while minimizing overall test cost. We now turn our attention to methods for combining the various algorithms and testers in ways that make it possible to achieve quality levels consistent with product requirements and design methodologies.

It has been recognized for some time now that true automatic test pattern generation is a long way from realization, meaning that software capable of automatically generating high-quality tests for most general sequential logic circuits does not currently exist, nor is it likely to exist in the forseeable future. Hence, it is necessary to incorporate testability structures in digital designs to make them testable.

We begin this chapter with a look at the design and test environment. That will provide a framework for discussion of the various topics related to test and will help us to see how the individual pieces fit together. Most importantly, by starting with a comprehensive overview of the total design and test process, we can identify opportunities to port test stimuli created during design verification into the manufacturing test development process. After examining the design and test environment, we will take an in-depth look at fault modeling because, in the final analysis, the fault model that is chosen will have a significant effect on the quality of the test. Other topics that fit into a comprehensive design and test framework, including design-for-test (DFT) and built-in-self-test (BIST), will be discussed in subsequent chapters.

7.2 THE TEST TRIAD

Several strategies exist for developing test programs for digital ICs; these include:

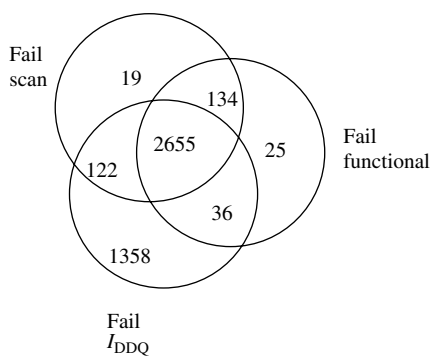
Functional vectors
Fault-directed vectors
I_{DDQ}

Digital Logic Testing and Simulation, *Second Edition*, by Alexander Miczo
ISBN 0-471-43995-9 Copyright © 2003 John Wiley & Sons, Inc.

Functional vectors may be derived from design verification suites or they may be written specifically to serve as manufacturing test programs. A fault simulator may be part of the selection/development process or the test program developer may take it on faith that his test program will effectively distinguish between faulty and fault-free product. Fault-directed vectors are usually generated by an automatic test pattern generator (ATPG), although the current state of the art in ATPG is quite primitive and commercial programs currently in existence operate either in full-scan or in partial-scan mode, where the percentage of storage devices (flip-flops and latches) in the scan path is usually in excess of 50% of the total number of storage devices. The I_{DDQ} test strategy (cf. Chapter 11) is based on the observation that CMOs circuits normally draw near-zero quiescent current when the clock is halted, and therefore defects in the form of shorts to ground or power will generate a quiescent current that is orders of magnitude greater than the normal quiescent current.

In a paper published in 1992, it was shown that a high-quality test benefited from all three of the test methodologies listed above.[1] The authors examined in detail a chip that contained 8577 gates and 436 flip-flops. A total of 26,415 die were analyzed. These were die that had passed initial continuity and parametric tests. Three different tests were applied to the die. The functional test had a coverage of 76.4% and the combined functional plus scan tests produced a combined stuck-at coverage of 99.3%.

Of the 26,415 die that were analyzed, 4349 were determined to be faulty. The Venn diagram in Figure 7.1 shows the distribution of failures detected by each of the three methods. Of the defective die, 2655 failed all three tests, 1358 die failed only the I_{DDQ} test, 25 die failed only the functional test, and 19 failed only the scan test, while 134 die failed both the functional and scan test, but passed the I_{DDQ} test. There were 122 die that failed I_{DDQ} and scan, but not the functional test, and 36 that failed I_{DDQ} and functional but not the scan test. For a product that requires the highest possible quality, the results suggest that tests with high stuck-at coverage and I_{DDQ} test are necessary. In this chapter we will focus on the functional test; in subsequent chapters we will examine in detail the scan, partial-scan, and I_{DDQ} test methodologies.



Distribution of failing die in each test class.

Figure 7.1 Results of different tests.

7.3 OVERVIEW OF THE DESIGN AND TEST PROCESS

A functional test program of the type referred to in the previous section can be derived as a byproduct of the design verification process. This section examines the design and test process, starting with the data flow diagram of Figure 7.2, which highlights the main features of a design and test workflow for an IC. The main features of the data flow diagram will be briefly described here; subsequent sections will cover the operations in greater detail. The *testbench* is a hardware design language (HDL) construct that instantiates a top-level module of a design whose correctness is being evaluated, together with additional software whose purpose is to stimulate the design and capture/print out response values. We assume in this discussion that the top-level circuit is an IC, rather than a PCB. We assume, further, that the circuit instantiated in the testbench is described using RTL (register transfer level) language constructs.

The testbench affords great flexibility in creating test stimuli for a design. The stimuli can be written in the same language as the circuit model, or in a special language perhaps better suited to describing waveforms to be applied to the circuit. The designer can incrementally add stimuli to the testbench and simulate until, at some point, he or she becomes convinced that circuit behavior conforms to some specification.

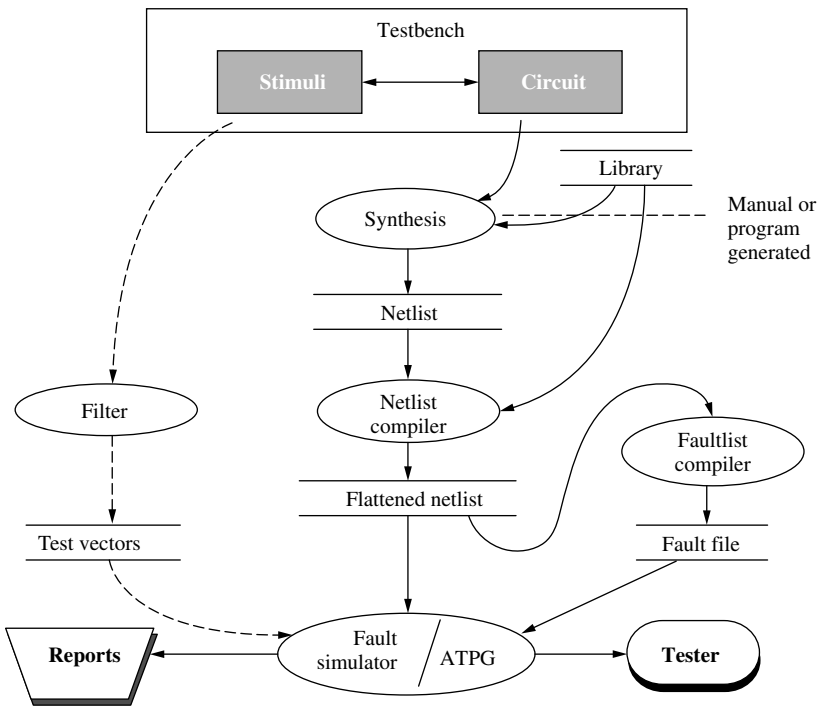


Figure 7.2 Design and test workflow.

At that point the design will be converted into a netlist. The conversion process can be performed manually or it can be accomplished through the use of synthesis programs. In practice, a typical IC will be synthesized using a combination of manual and automatic means. Some modules, including memories (RAM and ROM) and large data path functions, are often handcrafted. In addition, state machines, control paths, and other logic that are synthesized via synthesis programs may receive additional scrutiny from the logic designer if subsequent simulation or timing analysis reveals that timing constraints are not satisfied.

The synthesized netlist is usually partitioned along the same boundaries as the original circuit, with the original RTL modules now represented as an interconnection of *macrocells* or *standard cells*. The macrocells are low-level functions, ranging from simple buffers to full-adders and multiplexers. The netlist compiler flattens the netlist so that module boundaries become indistinguishable. However, naming conventions are used that make it possible to identify, hierarchically, where the logic element originated. For example, if top-level module *A* contains module *B*, and *B* contains an AND gate labeled *C*, then in the flattened netlist the AND gate could be recognized as *A.B.C*, or it could be recognized as *B.C*, where the top-level module *A* is implied; that is, every element is contained in the top-level module.

From the flattened netlist the fault-list compiler produces a fault file. The fault file is extremely important because it is used to measure the effectiveness of test programs. The fault-list compiler must create a fault list that is representative of faults in the circuit, but at the same time it must be careful to produce a fault list that can be simulated in a reasonable amount of CPU time. It is possible for the fault simulator to be extremely accurate and efficient, and still produce deceptive and/or meaningless results if the fault list that it is working from is not a representative fault list. Walking the tightrope between these sometimes conflicting requirements of accuracy and speed is a major challenge that will receive considerable attention in this chapter.

The fault simulator and ATPG algorithms received considerable attention in previous chapters. Here we simply note that, if a test strategy includes an ATPG, then the netlist must be expressed as an interconnection of primitives recognized by the ATPG. If the netlist includes primitives not recognized by the ATPG, these primitives must be remodeled in terms of other primitives for which the ATPG has processing capability. This is usually accomplished as part of the library development/maintenance task. A singular cover, propagation D-cubes, and primitive D-cubes of failure (PDCF) may also exist for circuit primitives, either in a library or built into the ATPG.

The purpose of the filter in Figure 7.2 is to select design verification vectors and reformat them for the target tester. By including a fault simulation operation in this phase of the task, it is possible to intelligently select a small subset of the design verification vectors that give acceptable fault coverage. This is necessary because design verification usually involves creation and simulation of far more vectors than could possibly fit into a tester's memory. More will be said about this in a subsequent section.

In this chapter, fault simulation and ATPG will be examined from the user's perspective. What kind of reports should be generated, and how do test programs get translated into tester format? Users have, in the past, been quite critical of fault simulators, complaining that they simply produced a fault coverage number based on the test vectors and the fault list, without producing any meaningful suggestions, help, or insight into how to improve on that number. We will examine ways in which fault simulation results can be made more meaningful to the end user.

The workflow depicted in Figure 7.2 is quite general; it could describe almost any design project. The circuit being designed may be constrained by rigid design rules or it may be free form, with the logic designers permitted complete freedom in how they go about implementing their design. However, as details get more specific (e.g., is the design synchronous or asynchronous?), choices start becoming bounded. Many of the vexing problems related to testing complex sequential circuits will be postponed to subsequent chapters where we address the issue of design-for-testability (DFT). For now, the focus will be on the fault simulator and the ATPG and how their interactions can be leveraged to produce a test program that is thorough while at the same time brief.

7.4 A TESTBENCH

A testbench will be created for the circuit in Figure 7.3 using Verilog. A VHDL description at the structural level would be quite similar, and the reader who understands the following discussion should have no difficulty understanding an equivalent VHDL description of this circuit. The testbench instantiates two modules; the first is the circuit description, while the second contains the test stimuli, including timing data. The circuit description is hierarchical, containing modules for a mux and a flip-flop. The test stimulus module follows the hierarchical netlist testbench.

7.4.1 The Circuit Description

The Verilog circuit description that follows is rather brief. The reader who wishes to acquire a more thorough understanding of the Verilog HDL is encouraged to consult

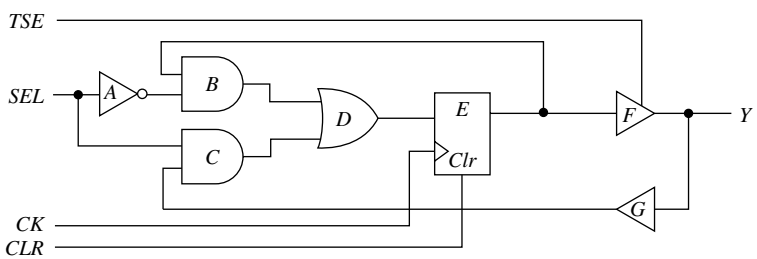


Figure 7.3 Gate-level interconnection.

one of the many textbooks dedicated to that subject. Because the language is quite robust, the following code represents but one of several ways to describe a particular behavior. Also note that the first line of each module is set in boldface for convenience in locating the start of each new module.

```
'timescale 1 ns / 100 ps
module testbench;
ckt7p3 X1 (tse, sel, ck, clr, y);
stimuli X2 (tse, sel, ck, clr, y);
endmodule
module ckt7p3 (tse, sel, ck, clr, y);
input tse, sel, ck, clr;
inout y;
wire hold;
wire load, choose;
mux2 x1 (.A(hold), .B(load), .Sel(sel), .C(choose));
dff x2 (.Q(hold),.QN(),.data(choose),.clock(ck),
.preset(1'b1),.clear(clr));
bufif1 #(7,7) x3 (y, hold, tse);
buf #(4,4) (load, y);
endmodule
module mux2(A, B, Sel, C);
input A, B, Sel;
output C;
not #(5,5) n1 (Sel_, Sel);
and #(5,5) n2 (L1, Sel_, A);
and #(5,5) n3 (L2, Sel, B);
or #(6,6) n4 (C, L1, L2);
endmodule
module dff(Q,QN,data,clock,preset,clear);
input data; input clock; input preset; input clear;
output Q;
output QN;
nand #(5,5)N1 (L1, preset,L4, L2),
  N2 (L2, L1, clear, clock),
  N3 (L3, L2, clock, L4), N4 (L4, L3, data, clear),
  N5 (Q, preset, L2, QN), N6 (QN, Q, L3, clear);
endmodule

module stimuli(tse, sel, ck, clr, y);
output tse, sel, ck, clr;
inout y;
```

```
reg [3:0] inputs;
reg ck;
parameter clock_high = 50; // 100ns period, clock high 50ns
'define cycle #1000 inputs = 4'b
assign {tse, sel, clr, y} = inputs;
initial begin
  ck = 0;
  $dumpfile("ckt7p3.dump");
  $dumpvars(3, X1);
  $monitor($time,," tse = %b sel = %b ck = %b clr = %b
              y = %b",
    tse, sel, ck, clr, y);
  'include "ckt7p3.fvc" // include vector file
  $finish; // end simulation
end
always #clock_high ck = ~ck;
endmodule

// ckt7p3.fvc -- tse, sel, clr, y
#0 inputs = 4'b110Z; // Reset
'cycle 0111; 'cycle 0111;
'cycle 101Z; 'cycle 101Z;
'cycle 110Z; 'cycle 111Z;
'cycle 0111; 'cycle 101Z;
'cycle 101Z; 'cycle 0110;
```

The first module in the listing is the top-level testbench, aptly named *testbench*. It begins with a timescale compiler directive that allows modules with different time units to be simulated together. The first number specifies the unit of measurement for delays in the module, and the second number specifies the accuracy with which delay values are rounded before being used in simulation. In the modules that follow, delays are multiples of 1 ns, and they are rounded to 100 ps during simulation. So, if a delay value of 2.75 is specified, it represents 2.75 ns and is rounded to 2.8 ns. The next entry is the name of the module, which ends with a semicolon, as do most lines in Verilog. The modules *ckt7p3* and *stimuli* are then instantiated. *Ckt7p3* contains the circuit description while the module *stimuli* contains the test program. Endmodule is a keyword denoting the end of the module.

The circuit *ckt7p3* again begins by listing the module name, followed by a declaration of the I/O ports in the circuit. The second line of ckt7p3 defines the ports *tse*, *sel*, *ck*, and *clr* as inputs. The third line defines the port *y* as an inout—that is, a bidirectional signal. The signals *hold*, *load*, and *choose* are internal signals. As wires, they can carry signals but have no persistence; that is, there is no assurance that values on those signals will be valid the next time the module is entered during simulation.

The next line instantiates *mux2*. It is a two-input multiplexer whose definition follows the definition for *ckt7p3*. Note that the signals in *mux2* are associated with wires in *ckt7p3* by using a period (.) followed by the signal name from *mux2* and then the wire called *hold* in *ckt7p3* is enclosed in parentheses. The signal named *Q* in *dff* is also associated with the wire *hold*. It is not necessary to associate names in this fashion, but it is less error-prone. If this method is not employed, then signals become position-dependent; in large circuits, errors caused by signals inadvertently juxtaposed can be extremely difficult to identify.

The *dff* instantiated in *ckt7p3* is the next module listed. It corresponds to the circuit in Figure 2.8. The signal 1'b1 connected to the preset in the *dff* denotes a logic 1. Similarly, 1'b0 denotes a logic 0. The next element in *ckt7p3* is called *bufif1*. The *bufif1* is a tri-state buffer and is a Verilog primitive. There is a corresponding element called *bufif0*. *Bufif1* is active when a logic 1 is present on its enable pin. *Bufif0* is active when the enable signal is a logic 0. Other Verilog primitives in the above listing include *buf*, *and*, *or*, and *nand*. Any Verilog simulator must provide simulation capability for the standard primitives.

Verilog does not support built-in sequential primitives for the latches and flip-flops; however, it does support user-defined primitives (UDPs). The UDP is defined by means of a truth table, and the facility for defining UDPs allows the user to extend the set of basic primitives supported by Verilog. Through the use of UDPs it is possible for the user to define any combination of gates as a primitive, so long as the model only contains a single output pin. Sequential elements can also be defined. The requirement is that the sequential element must directly drive the output.

7.4.2 The Test Stimulus Description

The module called *stimuli* has the same I/O ports as *ckt7p3*. However, in this module the signals that were inputs in *ckt7p3* have become outputs. The inout signal *y* remains an inout. A 4-bit register named *inputs* is defined. The "reg" denotes an abstract storage element that is used to propagate values to a part. The signal called *ck* is defined as a register. Then a parameter called *clock_high* is defined and set equal to 500. That is followed by the definition of the ASCII string #1000 inputs = 4'b. These two statements are used to define a clock period of 1000 ns, with a 50% duty cycle. The values in the register *inputs* are assigned to the input and inout signals by means of the assign statement that follows.

An initial statement appears after the assign statement. The first initialization statement causes a 0 to be assigned to *ck* prior to the start of simulation. Then a dump-file statement appears; it causes internal signal values to be written to a dump file during simulation. The dumpvars statement requests that the dump be performed through three levels of hierarchy. The dump file holds values generated by internal signals during simulation so that they can later be retrieved for visual waveform display.

In the ckt7p3 circuit, there are three levels of hierarchy; the top level contains *mux2* and *dff*, and these in turn contain lower-level primitive elements. The monitor statement requests that the simulator print out specified values during simulation so

that the user can determine whether the simulation was successful. It instructs the simulator on how to format the signal values. The text enclosed in quotes is the format statement; it is followed by a list of variables to be printed. The include statement requests that a file named ckt7p3.fvc be included; this file contains the stimuli to be simulated. The $finish indicates the end of simulation. The *ck* signal is assigned an initial value of 0. Then, every 500 ns it switches to the opposite state.

The next file contains the stimuli used during simulation. Although the stimuli in this example are vectors listed in matrix form, they could just as easily be generated by a Verilog model whose sole purpose is to emit stimuli at random times, thus imitating the behavior of a backplane. In this vector file, the word *cycle* is replaced by the ASCII text string defined in stimuli.v. That text contains a time stamp, set to the value 1000. The simulator applies each vector 1000 time units after the previous vector. The time stamp is followed by the variable *inputs*; it causes the following four values to be assigned to the variable *inputs* from which they will subsequently be assigned to the four I/O ports by the assign statement.

The values begin with the number 4, indicating the number of signal values in the string; the apostrophe and the letter b indicate that the string is to be interpreted as a set of binary signals. The four values follow, ended by a semicolon. The values are from the set {0, 1, X, Z}. The fourth value is applied to the inout signal *y*. Recall the *y* is an inout; sometimes it acts as an input, and other times it acts as an output. When *y* acts as an input, a logic 0 or 1 can be applied to that pin. When *y* acts as an output, then the I/O pad is being driven by the tri-state buffer, so the external signal must be a floating value; in effect the external driving signal is disconnected from the I/O pad.

7.5 FAULT MODELING

In Chapter 3 we introduced the basic concept of a stuck fault. That was followed by a discussion of equivalence and dominance. The purpose of equivalence and dominance was to identify stuck-at faults that could be eliminated from the fault list, in order to speed up fault simulation and test pattern generation, without jeopardizing the validity of the fault coverage estimate computed from the representative faults. Other factors that must be considered were postponed so that we could concentrate on the algorithms. The fault list is determined, at least in part, by the primitives appearing in the netlist. But, even within primitives, defects in different technologies do not always produce similar behavior, and there are several MOS and bipolar technologies in use.

7.5.1 Checkpoint Faults

Theorem 3.3 asserted that in a fanout-free circuit realized by symmetric, unate gates, it was sufficient to put SA1 and SA0 faults on each primary input. All of the interior faults are either equivalent to or dominate the faults on the primary inputs. All faults interior to the circuit will be detected if all the faults on the inputs are detected. This

suggests the following approach: identify all fanout-free regions. Start by identifying logic elements that drive two or more destination gates. That part of the wire common to all of the destination gate inputs is called a *stem*. The signal path that originates at a primary input or at one of the fanout paths from a stem is called a *checkpoint arc*.[2] Faults on the gate inputs connected to checkpoint arcs are called *checkpoint faults*.

It is possible to start out with a fault set consisting of SA0 and SA1 faults at all checkpoint arcs and stems. This set can be further reduced by observing that if two or more checkpoint arcs terminate at the same AND (OR) gate, then the SA0 (SA1) faults on those arcs are equivalent and all but one of them can be deleted from the fault list. The remaining SA0 (SA1) fault can be transferred to the output of the gate.

Example The circuit in Figure 7.4 has eight checkpoint arcs: four primary inputs and two fanout paths from each of *P* and *R*. Therefore, there are initially 16 faults. Faults on the inputs of the inverters can be transferred to their outputs; then the faults on the output of Q can be transferred to the input to S. The 16 faults now appear as SA0 and SA1 faults on the outputs of P and R and on each of the three inputs to S and T. The SA0 faults at the inputs of AND gates S and T are equivalent to a single SA0 fault on their outputs; hence they can be represented by equivalent SA0 faults, resulting in a total of 12 faults. ■■

Using checkpoint arcs made it somewhat simpler to algorithmically create a minimum or near minimum set of faults, in contrast to assigning stuck-at faults on all inputs and outputs of every gate and then attempting to identify and eliminate equivalent or dominant faults. In general, it is a nontrivial task to identify the absolute minimum fault set. Recall that fault b dominates fault a if $T_a \subseteq T_b$, where T_e is the set of all tests that detect fault e. If b is a stem fault and a is a fault on a checkpoint arc and is $T_a = T_b$, then fault b can be omitted from the fault list. But, consider the circuit of Figure 4.1. If the test vector $(I_1, I_2, I_3, I_4, I_5) = (0, 0, 1, 0, 0)$ is applied to the circuit, an SA0 on the output of gate D will not be detected, but an SA0 on the input to gate I driven by gate D will be detected, as will an SA0 on the input to inverter J (verify this).

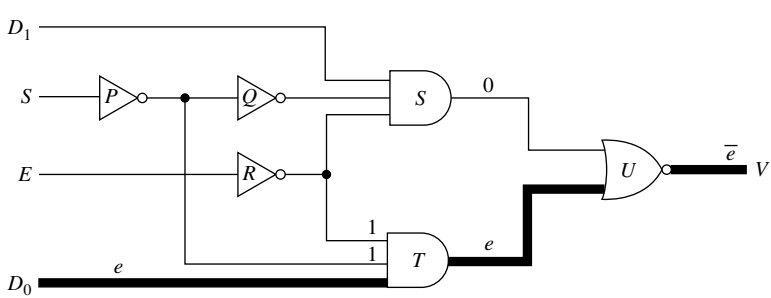


Figure 7.4 Propagating a signal.

Checkpoint faults can be associated with unique signal path fragments. This is illustrated in Figure 7.4. The bold lines identify a signal path from input D_0 to the output. During design verification it would be desirable to verify that the indicated path behaves as intended. Verification involves propagating a signal $e \in \{0,1\}$ from input D_0 to the output while all other signals are in an enabling state. But, there are many such signal path fragments. How can we be sure that all such paths have been verified?

Note that sensitization of the path is no more and no less than a sensitization of the SA1 on the input to gate *T* and an SA0 on the output of gate *T*. An SA1 on the input to *T* can only be detected if a logic 0 can be propagated from D_0 to the output *V* in such a way that the output value functionally depends on the presence or absence of the stated fault. Meanwhile, an SA0 on the output of *T* can only be detected if a 1 can be successfully propagated from D_0 to *V*. Hence, if tests can be created that detect both of those faults, then a test has been created that can serve as part of a design verification suite.

The point of this discussion is that if a test detects all stuck-at faults, then the test is also useful for verifying correctness of the design (note that it is necessary, of course, to verify circuit response to the stimuli). Conversely, if a design verification suite detects all checkpoint faults, then that suite is exercising all signal path fragments during times when they act as controlling entities—that is, when the circuit is conditioned such that an output is functionally dependent on the values being propagated. If the test does not detect all of the faults, then it is missing (i.e., not exercising), some signal path fragments. Hence, the fault coverage number is also a useful metric for computing thoroughness of a design verification suite.

7.5.2 Delay Faults

A circuit may be free of structural defects such as opens and shorts and yet produce incorrect response because propagation delay along one or more signal paths is excessive. Simply propagating 1 and 0 along these paths, while sufficient to detect stuck-at faults, is not sufficient to detect delay faults since the signal propagating to a flip-flop or primary output may have the same value as the previous signal. It cannot then be determined whether the signal clocked into the flip-flop or observed at a primary output is the new signal or the old signal.

Detecting delay faults requires propagating rising and falling edges along signal paths (cf. Section 3.8). The existence of checkpoint faults as identifiers of unique signal paths for propagation of 1 and 0 suggests the following strategy to detect both stuck-at faults and delay faults:

1. Identify all unique signal paths.
2. Select a path, apply a 0 to the input, then propagate through the entire path.
3. Repeat the signal propagation with a 1, and then again with a 0, on the input.
4. Continue until all signal paths have been exercised.

The test strategy just described will check delay relative to clock pulse duration along paths where source and destination may be flip-flops and/or I/O pins. The strategy is also effective for detecting stuck-open faults in CMOS circuits (see Section 7.6.3). The number of unique signal paths will usually be considerably less than the number of checkpoint faults since several faults will usually lie along a given signal path. Since the task of identifying signal paths and creating rising and falling edges can be compute-intensive, it may be advisable to identify signal paths most likely to have excessive delay and limit the propagation of edges to those paths.

Note that a complete signal path can include several flip-flops. It is not an easy task to set up and propagate rising and falling edges along all segments of such paths. For example, an ALU operation may be needed in a CPU to set up a 0 or 1. By the time the complementary value has been set up several state transitions later, the original value may have changed unintentionally. A concurrent fault simulator can be instrumented to identify and track edge faults, just as easily as it tracks stuck-at faults, and it can identify paths or path segments that have been exercised by rising or falling edges.

7.5.3 Redundant Faults

Redundant connections can cause a fault to be undetectable. A connection is defined as *redundant* if it can be cut without altering the output functions of a circuit.[3] If a circuit has no redundant connections, then it is *irredundant*. The following theorem follows directly from the definition of redundancy.

Theorem 7.1 All SA1 and SA0 faults in a combinational circuit are detectable iff the circuit is irredundant.

The simplest kind of redundancy, when discrete components are used, is to tie two or more signal pins together at the input of an AND gate or and OR gate. This is done when an n-input gate is available in an IC package and a particular application does not require all the inputs. For example, if an AND gate has inputs A, B, and C and if inputs A and B are tied together, then input combinations A, B, $C = (0,1,1)$ or $(1,0,1)$ are not possible. So SA1 faults on inputs A and B are undetectable.

Consider what happens when an open occurs on a net where two inputs are tied together (Figure 7.5). There are two possibilities:

1. An open occurs somewhere between the common connection point and one of the inputs.
2. An open occurs prior to the common connection point.

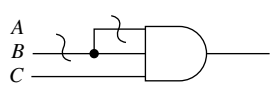


Figure 7.5 AND gate with redundant input.

If an open exists between the common connection and the gate input, then the fault cannot be detected. If an open occurs prior to the common connection of the inputs, then the open affects both inputs and circuit behavior is the same as if there were a single input with a SA1 on the input.

The redundancy just described is easily spotted simply by checking for identical names in the gate input list. If matching signal names are found, then all but one signal can be deleted. Other kinds of redundancy can be more difficult to detect. Redundancy incorporated into logic to prevent a hazard will create an undetectable fault. If the fault occurs, it may or it may not produce an error symptom since a hazard represents only the possibility of a spurious signal. No general method exists for spotting redundancies in logic circuits.

7.5.4 Bridging Faults

Faults can be caused by shorts or opens. In TTL logic, an open at an input to an AND gate prevents that input from pulling the gate down to 0; hence the input is SA1. Shorts can be more difficult to characterize. If a signal line is shorted to ground or to a voltage source, it can be modeled as SA0 or SA1, but signal lines can also be shorted to each other. In any reasonably sized circuit, it is impractical to model all pairs of shorted nets. However, it is possible to identify and model shorts that have a high probability of occurrence.

Adjacent Pin Shorts A function F is *elementary* in variable x if it can be expressed in the form

$$F = x^* \cdot F_1$$

or

$$F = x^* + F_2$$

where x^* represents x or $\bar{x}$ and F_1, F_2 are independent of x. An *elementary gate* is a logic gate whose function is elementary. An *input-bridging fault* of an elementary gate is a bridging fault between two gates, neither of which fans out to another circuit. With these definitions, we have:[4]

Theorem 7.2 A test set that detects all single input stuck-at faults on an elementary gate also detects all input-bridging faults at the gate.

The theorem states that tests for stuck-at faults on inputs to elementary gates, such as AND gates and OR gates, will detect many of the adjacent pin shorts that can occur. However, because of the unpredictable nature of pin assignment in IC packages (relative to test strategies), the theorem rarely applies to IC packages. It is common in industry to model shorts between adjacent pins on these packages because shorts have a high probability of occurrence, due to the manufacturing methods used to solder ICs to printed circuit boards.

Adjacent pin shorts may cause a signal on a pin to alter the value present on the other pin. To test for the presence of such faults, it is necessary to establish a sensitized signal on one pin and establish a signal on the other pin that will pull the sensitized pin to the failing value. If the sensitized value D ($\overline{D}$) is established on one of the pins, then a 0 (1) is required on the adjacent pin. Given a pair of pins P_1 and P_2, the following signal combinations will completely test for all possibilities wherein one pin may pull another to a 1 or 0.

P_1:	D	$\overline{D}$	0	1
P_2:	0	1	D	$\overline{D}$

It is possible to take advantage of an existing test to create, at the same time, a test for adjacent pin shorts. If a path is sensitized from an input pin to an output pin during test pattern generation and if a pin adjacent to the input pin has an x value assigned, then that x value can be converted to a 1 or 0 to test for an adjacent pin short. The value chosen will depend on whether the pin on the sensitized path has a D or $\overline{D}$.

Programmable Logic Arrays Shorts created by commercial soldering techniques are easily modeled because the necessary physical information is available. Recall that IC models are stored in a library and are described as an interconnection of primitives. That same library entry can identify the I/O pins most susceptible to solder shorts, namely, the pins that are adjacent.

Structural information is also available for programmable logic arrays (PLAs) and can be used to derive tests for faults with a high probability of occurrence. Logically, the PLA is a pair of arrays, the AND array and the OR array. The upper array in Figure 7.6 is the AND array. Each vertical line selects a subset of the input variables, as indicated by dots at the intersections or crosspoints, to create a product term. The lower array is the OR array. Each horizontal line selects a subset of the product terms, again indicated by dots, to create a sum-of-products term at the outputs.

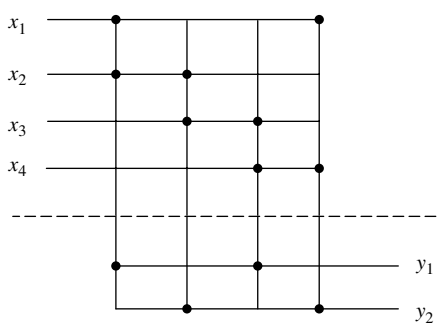


Figure 7.6 Programmable logic array.

The PLA is susceptible to bridging faults and crosspoint faults.[5] The *crosspoint fault* is a physical defect caused by a diode at a crosspoint that is connected (unconnected) when it should not (should) have been connected. In the AND array, the product term logically shrinks if a device is disconnected and the product term logically expands if an additional input variable is connected to the vertical line. In the OR array, a product term is added if an additional column is connected into the circuit, and a product term will disappear from the circuit output if a column is not connected where required.

Bridging faults can occur where lines cross. The symptom is not necessarily the same as when an additional device is connected into a circuit. For example, the bridging fault may cause an AND operation, whereas the crosspoint fault may cause an OR operation. The crosspoint open is similar in behavior to opens in conventional gates. The bridging fault, like shorts between signal lines in any logic, is complicated by the fact that a signal is affected by a logically unrelated signal. However, the regular structure of the PLA makes it possible to identify potential sources of bridging faults and to perform fault simulation, if necessary, to determine which of the possible bridging faults are detected by a given set of test patterns.

7.5.5 Manufacturing Faults

Creation of test stimuli and their validation through fault simulation can be a very CPU-intensive activity. Therefore, when testing PCBs it has been the practice to direct test pattern generation and fault simulation at fault classes that have the highest probability of occurrence. In the PCB environment, two major fault classes include *manufacturing faults* and *field faults*. Manufacturing faults are those that occur during the manufacturing process, and include shorts between pins and opens between pins and runs on the PCB. Field faults occur during service and include opens occurring at IC pins while the IC is in service, but also include internal shorts and opens.

Testing in a manufacturing environment is often restricted to manufacturing faults because it is assumed that individual ICs have been thoroughly tested for internal faults before being mounted on the board. Although this can significantly reduce CPU time, the test so generated suffers from the drawback that it may be inadequate for detecting faults that occur while the device is in service. Studies of fault coverage conducted many years ago on PCBs comprised mainly of SSI and MSI parts showed that tests providing coverage for about 95% of the manufacturing faults often provided only about 70–75% coverage for field faults.[6,7] This problem of granularity has only gotten worse as orders of magnitude more logic is integrated onto packages with proportionately fewer additional pins.

7.6 TECHNOLOGY-RELATED FAULTS

The effectiveness of the stuck-at fault model has been the subject of heated debate for many years. Some faults are technology-dependent and cause behavior unlike

the traditional stuck-at faults. Circuits are modeled with the commonly used logic symbols in order to convey a sense of their behavior, but in practice it is quite difficult to correlate faults in the actual circuit with faults in the behaviorally equivalent circuit represented by logic gates. This is particularly true of faults that cause feedback (i.e., memory), in a combinational circuit.

7.6.1 MOS

A metal oxide semiconductor (MOS) circuit can also be implemented in ways that make it difficult to characterize faults. The circuit of Figure 7.7 is designed to implement the function

$$F = (A + C)(B + D)$$

With the indicated open it implements

$$F = A \cdot B + C \cdot D$$

It is not immediately obvious how to implement this MOS circuit as an interconnection of logic gates so as to conveniently represent both the fault-free and faulted versions (although it can be done).

7.6.2 CMOS

The complementary metal oxide semiconductor (CMOS) NOR circuit is illustrated in Figure 7.8. When A and B are low, both p-channel transistors are on, and both n-channel transistors are off. This causes the output to go high. If either A or B goes high, the corresponding upper transistor(s) is cut off, the corresponding lower transistor(s) is turned on, and the output goes low.

Conventional stuck-at faults occur when an input or output of a NOR circuit shorts to V_{SS} or V_{DD} or when opens occur at the input terminals. Opens can cause SA1 faults on the inputs because the input signal cannot turn off the corresponding

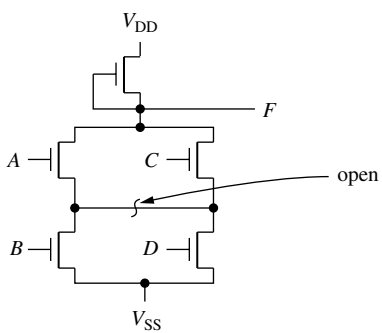


Figure 7.7 MOS circuit with open.

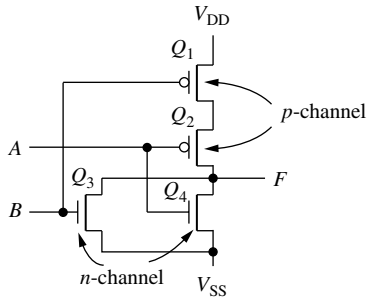


Figure 7.8 CMOS circuit.

p-channel transistor and cannot turn on the corresponding *n*-channel transistor. Opens can also occur in a transistor or at the connection to a transistor. Three such faults can be identified in the two-input NOR gate of Figure 7.8. These faults, usually referred to as *stuck-open* faults, include a defective pulldown transistor connected to A or B or an open pullup transistor anywhere between the output channel and V_{DD}.[8]

If Q_4 is open, a logic 1 at A can cut off the path to V_{DD} but it cannot turn on the path to V_{SS}. Therefore, the value at F will depend on the electrical charge trapped at that point when signal A goes high. The equation for the faulted circuit is

$$F_{n+1} = \overline{A}_{n+1} \cdot \overline{B}_{n+1} + A_n \cdot \overline{B}_n \cdot F_n$$

Table 7.1 illustrates the effect of all seven faults. In this table, F represents the fault-free circuit. F_1 and F_2 represent the output SA0 and SA1, respectively. F_3 and F_4 represent open inputs at A and B. F_5 and F_6 correspond to opens in the pulldown transistors connected to A or B or the leads connected to them. F_7 is the function corresponding to an open anywhere in the pullup circuit.

Some circuit output values become dependent on previous values held by circuit elements when the circuit is faulted, so that in effect the faulted circuit exhibits sequential circuit behavior. For example, note from Table 7.1 that F_5 differs from F, the fault-free circuit, only in row 3, and then only when F has value 0 and F_5 had a 1 at the output on the previous pattern. To detect this fault, it is necessary to establish the values (0, 0) on the inputs A and B. This produces the value 1 at the output of the gate.Then, the values (1, 0) are applied to the inputs and the sensitized value is propagated to an output.

TABLE 7.1 Fault Behavior for CMOS NOR

A	B	F	F_1	F_2	F_3	F_4	F_5	F_6	F_7
0	0	1	0	1	1	1	1	1	F_n
0	1	0	0	1	0	1	0	F_n	0
1	0	0	0	1	1	0	F_n	0	0
1	1	0	0	1	0	0	0	0	0

A suggested approach for testing stuck-open faults[9] develops tests for the traditional stuck-at faults first. When simulating faults, the previous pattern is checked to see if the value F_n from the previous pattern, in conjunction with the present value, will cause the output of the gate to be sensitized on the present pattern. In the situation cited in the previous paragraph, if the previous pattern causes a (0,0) to appear on the inputs of the NOR, and if the present pattern applies a (0,1) or (1,0) to the NOR, then one of the two stuck-opens on the pull down transistors is sensitized at the output of the NOR and it simply remains to simulate it to determine if it is sensitized to an output.

If stuck-open faults remain undetected after all stuck-at faults have been processed, it becomes necessary to explicitly sensitize them using a two-pattern sequence. The first pattern need only set up the initial conditions on the gate being tested. The second pattern must cause an error signal to be propagated to an output. Note that when simulating these patterns, it is also possible to check for detection of other stuck-open faults. CMOS library models may be too complex to process by comparing past and present values on input pins. It may be necessary to perform a switch-level fault simulation to determine if an input combination sensitizes a particular transistor open. As pointed out in Section 2.10, channel connected components can be simulated at the switch level and, if the output differs from the fault-free component, a fault effect can be diverged as a unidirectional element by a concurrent fault simulator.

7.6.3 Fault Coverage Results in Equivalent Circuits

The preceding examples illustrate the problems that exist when digital circuits are modeled at the gate level. In another investigation, this one involving emitter-coupled logic (ECL), a macro-cell library that included functions at the complexity of full-adders was examined. The authors demonstrated a need for test patterns over and above those that gave 100% coverage of the stuck-at faults for the gate-equivalent model.[10] Wadsack identified a similar situation wherein a small CMOS circuit had 100% stuck-at coverage and yet, on the tester, devices were failing on vectors after the point where 100% stuck-at coverage had occurred.[11]

It is simply not possible to represent a large ensemble of transistors as a collection of gates and expect to obtain a perfect test for the transistor level circuit by creating tests for the gate equivalent model. The larger the ensemble, the more difficult the challenge. Recall the observation made in Chapter 1: Testing is as much an economic challenge as it is a technical challenge. The ideal technical solution is to perform fault simulation at the transistor level. That, however, is not economically feasible.

To see just how difficult the problem of modeling circuit behavior can be, consider the rather simple circuit represented in Figure 7.9 as a sum of products and as a product of sums. These circuits are logically indistinguishable from one another, except possibly for timing variations, when analyzed at the terminals. However, the set of six vectors listed below will test all SA1 and SA0 faults in the NAND model but only 50% of the faults in the NOR model. In fact, two of the NOR gates could be completely missing and the test set would not discover it![12]

	Test Set			
	x_1	x_2	x_3	x_4
1:	1	1	1	1
2:	0	0	0	0
3:	1	0	0	0
4:	0	1	0	0
5:	0	0	1	0
6:	0	0	0	1

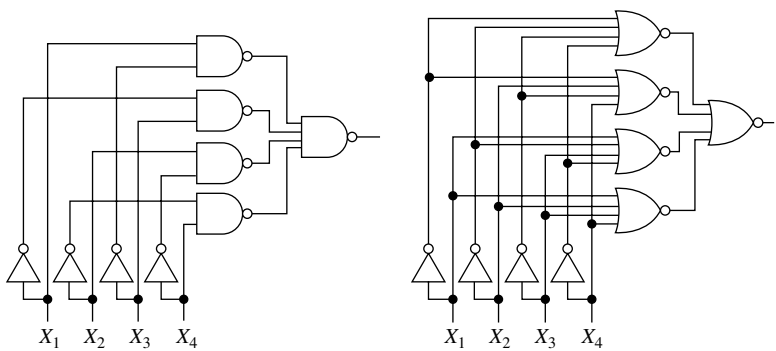


Figure 7.9 Two equivalent circuits.

Fortunately, circuits in real life are rarely that small. Fault coverage for structurally equivalent circuits generally tends to converge as it approaches 100%. This can be interpreted to mean that if your coverage for the gate equivalent circuit is 70%, it doesn't matter whether the real fault coverage is 68% or 72%, you can be reasonably confident that many faulty devices will slip through the test process. If your coverage is computed to be 99.9%, the real coverage may be 99.7% or 99.94%. In either case you will have significantly fewer tester escapes than when the fault simulator predicts 70% coverage. Fault simulation results, while not exact, can set realistic expectations with respect to product defect levels.

7.7 THE FAULT SIMULATOR

Although there is a growing trend toward DFT as circuits continue to grow larger, there still remain many products that are small enough to be adequately tested using vectors generated either during design verification or manually as part of a targeted test vector generation process. In this section we will discuss some features and

attributes of fault simulation that will enable a user to design strategies that are more productive, irrespective of whether or not an ATPG is employed.

7.7.1 Random Patterns

The use of random patterns is motivated by the efficiency curve shown in Figure 7.10. The first dozen or so patterns applied to a combinational logic circuit typically detect anywhere from 35% to 60% of the faults selected for testing, after which the rate of detection falls off.

To see why this curve holds, consider that any of 2^{2^n} functions can be implemented by a simple n-input, 1-output circuit. Any single test pattern in which all inputs have known values, 0 or 1, will partition the functions into two equivalence classes, based on whether the output response is a 1 or 0. The response of half the functions will match the response of the correct circuit. A second input will further partition the functions so that there are four equivalence classes. The functions in three of the classes will disagree with the correct circuit in one or both of the output responses. In general, for a combinational circuit with n inputs, and assuming all inputs are assigned a 1 or 0, the percentage of functions distinguished from the correct function after m patterns, $m < 2^n$, is given by the following formula:

$$P_D = \left(\frac{1}{2^{2^n} - 1} \cdot \sum_{i=1}^{m} 2^{2^n - i} \right) \cdot 100\%$$

The object of a test is to partition functions into equivalence classes so that the fault-free circuit is in a singleton set relative to functions that represent faults of interest. Since a complete partition of all functions is usually impractical, a fault model, such as the stuck-at model, defines the subset of interest so that the only functions in the equivalence class with the fault-free circuit are functions corresponding to faults with low probability of occurrence. A diagnostic test can also be defined in terms of partitions; it attempts to partition the set of functions so that as many functions as practical, representing faults with high probability of occurrence, are in singleton sets.

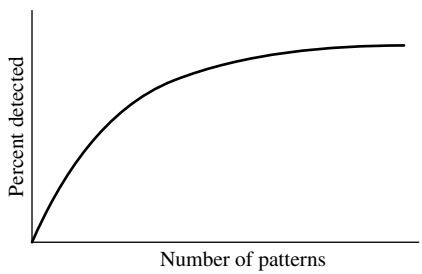


Figure 7.10 Test efficiency curve.

Example The 16 possible functions that can be represented by a two-input circuit are listed below. The two-input EXOR circuit is represented by F_6. Its output is 1 whenever A and B differ.

A	B	F_0	F_1	F_2	F_3	F_4	F_5	F_6	F_7	F_8	F_9	F_{10}	F_{11}	F_{12}	F_{13}	F_{14}	F_{15}
0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1
0	1	0	0	0	0	1	1	1	1	0	0	0	0	1	1	1	1
1	0	0	0	1	1	0	0	1	1	0	0	1	1	0	0	1	1
1	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1

■■

Application of any single pattern to inputs A and B distinguishes between F_6 and eight of the other 15 functions. Application of a second pattern will further distinguish F_6 from another four functions. Hence, after two patterns, the correct function is distinguished from 80% of the possible functions. The formula expresses percentage tested for these single-output combinational functions strictly on the basis of the number of unique input patterns applied and makes no distinction concerning the values assigned to the inputs. It is a measure of test effectiveness for all kinds of faults, single and multiple, and suggests why there is a high initial percentage of faults detected.

However, the formula does not provide any information about particular classes of faults, and, in fact, simulation of single stuck-at faults generally reveals a somewhat slower rise in percent of faults detected. This should not be surprising, however, since there are many more multiple faults than single faults and there is no evidence to suggest that detection of single and multiple faults occurs at the same rate. As pointed out earlier in this chapter, detection rates between manufacturing and field faults differs significantly.

Random patterns are significantly less effective when applied to sequential circuits. They are also ineffective, after the first few patterns, against certain fault classes with high probability of occurrence, such as stuck-at faults in combinational circuits. At that point the problem has shifted. Initially, the goal is to detect large numbers of faults. Then, after reaching some threshold, the goal is to detect specific faults. When random patterns are employed, their use is normally followed by deterministic calculation of test patterns for specific faults.

7.7.2 Seed Vectors

Random vectors are quite useful in combinational circuits. However, sequential circuits with tens or hundreds of thousands of logic gates and numerous complex state machines engaged in extremely detailed and sometimes lengthy "hand-shaking" sequences tend to be quite *random-resistant,* meaning that sequences of input stimuli applied to the circuit must be precisely calculated to steer the circuit through state transitions. Any single misstep in a sequence of n vectors can frustrate attempts to reach a desired state. Logic designers frequently spend considerable amounts of

time developing test sequences whose purpose is to steer a design through carefully calculated state transitions in order to check out and verify that the design is correct. These vector sequences, captured from a testbench, can often be used to advantage as part of a manufacturing test or as a framework for developing a more comprehensive manufacturing test.

Consider, again, the test triad discussed at the beginning of this chapter. It was pointed out that a comprehensive and effective test strategy can benefit from a functional test even in those instances where a high-fault-coverage test is generated by a full-scan-based ATPG. The functional vectors can be derived from the testbench used for design verification. With effective fault management tools the faults detected by the functional test sequences can be deleted from the fault list and the ATPG can focus its attention on those faults that escaped detection by the functional test vectors.

Capturing test vectors requires answering two questions: How are the test vectors to be captured and, after capturing them, which vector sequences should be kept? In a typical testbench, the sequences of vectors applied to the design may employ extremely complex timing. During a single clock period, numerous vectors may be generated by the testbench and applied at random intervals to the design. Furthermore, the design may have many bidirectional pins that are constantly switching mode, some acting as inputs and others acting as outputs. If these sequences of vectors are to be ported to a tester, they must conform to the tester's architectural constraints.

The tester will have a finite, limited amount of memory while the testbench may be generating stimuli randomly, pseudo-randomly or algorithmically during each clock period. Furthermore, many of the sequences created by the testbench may be repetitive and may not be contributing to overall fault coverage. By contrast, within the confines of the limited amount of tester memory it is desirable to store, and apply to the design, a test program that is both efficient and effective. The tester is an expensive piece of hardware; if the test program that is being applied to the IC is ineffective, then the user of that tester is not getting a reasonable return on investment (ROI).

Capturing Design Verification Vectors A testbench used in conjunction with an HDL model can be quite simple. It might simply be an array of vectors applied, in sequence, to the target device. Alternatively, the testbench may be a complex behavioral model whose purpose is to emulate the environment in which the design eventually operates. In the former case, it is a simple matter to format the array of vectors and input them to a fault simulator as depicted in Figure 7.2. Many sequences of vectors can be sent through the fault simulator and evaluated, with those most effective at improving fault coverage retained and formatted for the tester. Because fault simulation is a compute-intensive activity, the task of evaluating design verification suites can be accomplished more quickly through the use of fault sampling (discussed in Section 7.7.3).

When a design verification suite is generated by a complex bus functional model (BFM) or similar such behavioral entity, with signals emanating from the stimulus generator at seemingly random times during each clock cycle, and converging on a

design that contains numerous bidirectional pins, the task of selecting vector suites and formatting them for the tester becomes a bit more involved. Referring again to Figure 7.2, code can be inserted in the testbench to sample stimuli arriving at the circuit from the stimulus generator. The criteria for selecting stimuli may include capturing stimuli at the I/O pads of the circuit under test whenever a clock edge occurs. The stimuli are then written to a file that can be evaluated via fault simulation, with the more effective stimuli formatted and ported to the tester.

One problem that must be addressed is signal direction on bidirectional pins. An I/O pad may be driven by the stimulus generator, or it may be driven by the circuit under test. This requires that enable signals on tri-state drivers be monitored. If the enable signal is active, then the bidirectional pin is being driven by the circuit under test. In that case, the vector file being created by the capture code in the testbench must insert a Z in the vector file. The Z represents high impedance; that is, the tester, and, consequently, the fault simulator, is disconnected from that pin so as not to create a conflict. This is illustrated in Figure 7.11. The external driver, in this case the vector file being generated in the testbench, will drive the I/O pad at some times, and at other times the internal logic of the IC will drive the pad. When the internal logic is driving the pad, the external signal must be inactive.

The circuit in Figure 7.3 and described in Section 7.4.1 illustrates the issues discussed here. It has four inputs and a bidirectional pin. The bidirectional pin sometimes acts as an output, in which case the externally applied signal must be Z. At other times the pin is used to load the register, so it acts as an input. At that time, the enable on the tri-state driver must not be active.

A potential problem when capturing stimuli at I/O pads is inadequate setup time. If signals at I/O pads are captured at the same time that a clock edge occurs, then data signal changes will occur simultaneous with the occurrence of clock edges. To resolve this the tester and the fault simulator must reshape the clock by delaying it sufficiently to satisfy setup time requirements. This is illustrated in Figure 7.12 where the original clock signal, CLK, is reshaped using timing sets (TSETs) on the tester. The rising edge can be delayed an arbitrary amount through use of the TSETs. A rather simple way to accomplish this is to request, via the TSET, that the clock signal be the complement of the value contained in the tester memory for the duration specified. Then, at the end of the elapsed period, CLK assumes the value contained in pin memory.

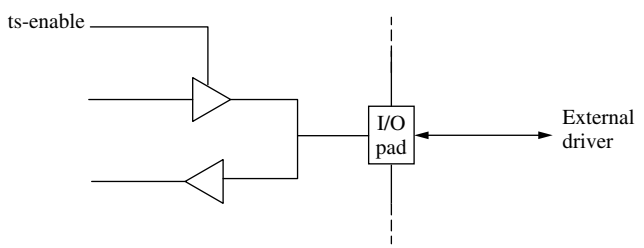


Figure 7.11 Bidirectional I/O pad.

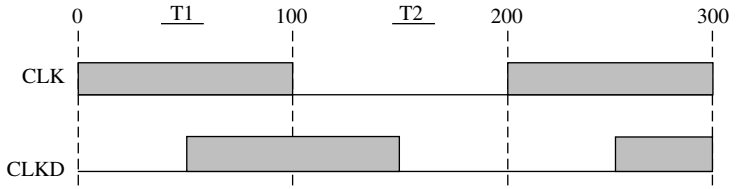


Figure 7.12 Shifting the clock edge.

Determining Which Vectors to Retain A typical design verification effort may generate many millions of test sequences, far more than could possibly fit into a typical tester memory. To select from these sequences a subset that provides good coverage of physical defects in the design requires fault simulation. But, fault simulation is a CPU intensive task. To perform a detailed fault simulation of all the design verification suites can take an incredibly long time. To assist in the selection process, two approaches can be employed: fault sampling and fault coverage profiles. We will now discuss each of these concepts in detail.

7.7.3 Fault Sampling

When a circuit is modeled at the gate level, the size of the fault list for that circuit, after collapsing, is generally in the range of 2.5X, where X is the number of logic gates. So, for example, a 100,000 gate circuit can be expected to have about 250,000 stuck-at faults in its fault list. If the objective is to sift through a large number of design verification vector suites in order to find a subset that provides useful fault coverage, then it is unnecessary to fault simulate the entire list of faults.

The practice of sampling can be put to good use in fault simulation. The object is to evaluate the effectiveness of one or more sets of test vectors with the smallest possible expenditure of CPU time, subject to the availability of main memory. When designers are generating many hundreds or thousands of test programs, often simulating them on specialized hardware simulators or emulators, over a period of several months, it is not practical to fault simulate all of the sequences in detail.

Fault sampling selects a subset of a total fault population for consideration during fault simulation. The goal is to quickly get a reasonably accurate estimate of the fault coverage produced by a set of test vectors. We consider here the development provided by Wadsack.[13] Consider a population of N faults and a test that can detect m of those faults. Assume that n out of N faults will be simulated. Let $f = m/N$ and $F = X/n$, where X is the number of faults detected from the random sample. Then f is the actual fault coverage and F is an approximation of f based on the sample. The variance of F is shown to be

$$Var(F) = (1 - n/N) \cdot f \cdot (1 - f) \cdot (1/n)$$

A 95% confidence level is twice the square root of the variance, so $f = F \pm 2(\mathrm{Var}(F))^{1/2}$. The graph in Figure 7.13 shows the variance for a 10% sample when $N = 100{,}000$. This graph reveals that the fractional error Z is likely to be less than 1%. Furthermore, the error is greatest at a coverage of 50% and approaches 0 as the fault coverage approaches 100%.

7.7.4 Fault-List Partitioning

Fault simulation can be extremely memory intensive, particularly when event-driven, full-timing, concurrent fault simulation is being performed on a large circuit. It is often the case that complete fault simulation of an entire fault set for large circuits simply is not possible due to insufficient memory. In such cases, the set of faults can be partitioned into several smaller sets and each fault set can be simulated individually. The results can be used to update a master fault list. If a fault list is partitioned into, say, 10 subsets, each containing 10% of the faults from a master fault list, then 10 passes will be required to completely fault simulate all of the subsets. If each of the subsets is a pseudo-random selection of faults, without replacement, from the master fault list, then the fault coverage percentage from each of these partitions should be approximately the same, as discussed in the preceding subsection. If the fault partition is made up of faults, all selected from the same functional area of the IC, then the fault coverage from these partitions can show substantial variation.

Fault partition sizes can be determined by the fault simulator. The operating system can advise as to how much memory is available to keep track of fault effects. The size of the data structure used to record fault effects is known and, with experience, a reasonably accurate estimate can be made of the number of fault effects that exist, on average, for each fault origin. With this information, it is possible to estimate how many faults can be processed in each fault simulation pass. If the estimate is too optimistic, and not enough memory exists to process all of the faults, then some of the faults can be deleted and fault simulation can continue with the reduced fault list. Those faults that were deleted can be added back in a subsequent fault partition.

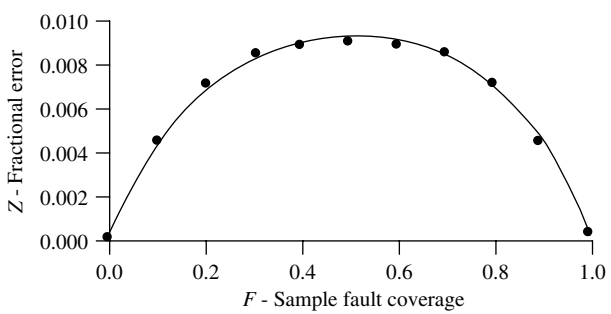


Figure 7.13 Ninety-five percent confidence interval.

7.7.5 Distributed Fault Simulation

Distributed fault simulation can be part of a comprehensive strategy in which the initial goal is to find a set of test programs that achieve high fault coverage, using fault sampling techniques. After there is some degree of confidence that the test programs produce high coverage, then a complete fault simulation of all faults from a master fault list can be performed, and the results can then be gathered up by the control program. If, at this point, the fault coverage is still marginally below that level needed to achieve a corporate AQL (acceptable quality level), then additional test programs, or perhaps some DFT, can be used to reach the target fault coverage level. In fact, this may be a critical juncture at which to make a decision as to whether or not the use of design verification vectors should be abandoned and replaced with a different test strategy, such as a full DFT. The decision might be made because the coverage goals cannot be achieved otherwise, or the decision might be made because the cost of testing each chip (time on the tester) may be too great.

When a fault list is partitioned, individual partitions can be run serially, on the same workstation, or they can be run in parallel over a network. A control program running on a master workstation can spawn subordinate processes on other workstations connected via the network. When these subordinate processes finish, they report their results to the control program, and the results are used to update a master fault list. These subordinate processes can be run as background tasks with low priority so that if a user is working interactively on a workstation, for example, editing a file, the subordinate process will not interfere with his or her activities.

7.7.6 Iterative Fault Simulation

During design verification, a common practice is to generate multiple files of stimuli. Each such file will be targeted at a specific area of the design, and these files may be created by different designers. There is often overlap between these files. If these files are to be used as part of the test program, then a common practice is to iterate through these files and determine how much coverage is provided by each of the design verification suites. With a large number of these design verification suites, it is not uncommon to see that some suites will provide significant coverage, while others may provide either very little coverage or perhaps no additional coverage.

If some suites provide very little coverage, then a decision must be made as to whether or not the use of those suites is justified. Their contribution to overall improvement in AQL may be negligible, while the test may contain so many vectors as to add a significant amount of time on the tester. A strategy that may prove useful is to fault simulate all of the design verification suites with a sample, say 10%, of the fault set. Toss out the suites that provide no additional coverage, then rank the remaining suites based on how much fault coverage they contribute to the total and resimulate. Some of the suites that had very low coverage during the first iteration may now drop out completely. This is essentially a covering operation, and it does not improve the fault coverage; the same faults will be detected, assuming the same fault sample is used, but the objective is to find the smallest set of suites that achieve

that fault coverage, hence the smallest number of vectors, thus reducing the amount of time the device spends on the tester.

7.7.7 Incremental Fault Simulation

Incremental fault simulation permits the user to conditionally create and apply stimuli to the circuit. These stimuli may be experimental. For example, the user may be trying to drive the circuit into a particular state in order to sensitize a group of faults that would otherwise go undetected. In order to achieve the goal, the user must be able to apply the stimuli and monitor response, including internal states of the circuit. In the event that stimuli do not achieve their desired end, it is also necessary, to be able to delete some or all of the stimuli. This implies an ability to checkpoint the circuit, and to back up to that checkpoint if analysis of simulation results identifies incorrect state transitions or some other reason for failure to improve fault coverage.

7.7.8 Circuit Initialization

Indeterminate states at the beginning of a simulation present a significant problem for fault simulators. Some designs, in particular those that take advantage of DFT structures, are able to initialize some or all of the circuit storage elements quite quickly, often simply by toggling a reset input. However, there are circuits that require complex sequences to drive all of the flip-flops and latches into a known state. Many fault detections during this initialization period are probable detects, in which the good circuit has a known value $e \in \{0,1\}$, and the faulty circuit has an unknown value, X. This composite signal e/X may propagate to an output where it is recorded as a probable detect. In this case, the response for the fault-free circuit is known, but the response for the faulty circuit has, on average, only a 50% probability of possessing a binary value that is different from the good circuit. A problem with probable detects is the fact that many applications require absolute detections, particularly in products where health or public safety are at risk. The probable detect may cause the fault simulator to ignore later absolute fault detects, thus obscuring the true fault coverage.

One way to deal with this is to simply ignore faults detected at the I/O pins until initialization is complete. However, this does not resolve the problem of probable detects. Suppose a reset input on a flip-flop is stuck to the inactive state. Then, in a concurrent fault simulator, the fault origin will spawn fault effects (cf. Section 3.7.2) that will reach an I/O pin, where they will be ignored until the fault simulator is told to begin recording detected faults.

An alternate approach is for the fault simulator to be configured to postpone propagation of fault effects until the circuit has reached a known state. Then, after the circuit has been initialized, if a flip-flop output switches from 0 to 1 (1 to 0), and if that transition causes a transition on an output, then a fault on, for example, the clock line would prevent the transition from occurring, and the observable signal would appear stable at the output when it should be switching. Thus, faults can be detected with certainty. In this arrangement it is possible that faults may actually be

detected sooner on the tester. But they could only be recorded as a probable detect by the fault simulator. This strategy requires the user to create an initialization sequence that fully initializes the circuit.

An alternate strategy for getting a full and accurate tabulation of faults that are absolute detects, and those that are only probable detects, is to run fault simulation twice. During the first run, fault simulation is configured to count only absolute detections. Then, on a final run, fault simulation is run with all the undetected faults, but it is configured to count probable detects. It may then be possible to set a threshold, requiring that a fault be counted as a probable detect if it is detected some minimum number of times. In commercial products, a default of five or ten probable detects is often set as a default.

7.7.9 Fault Coverage Profiles

For many years, fault simulation simply consisted of generating lists of faults, collapsing the lists, and then running one or more files of test vectors against the netlist and fault list to determine fault coverage provided by the set(s) of test vectors. If fault coverage was satisfactory, their job was done. But, if fault coverage was unsatisfactory, engineers writing additional test vectors to improve fault coverage frequently would work in the blind. It was possible to get a list of detected and undetected faults, but the data were simply too overwhelming to be of any value. The fault coverage *profiler*, or reporter, as it is sometimes called, is a data reduction tool. It enables the user to generate detailed reports on fault coverage.

An overall fault coverage of 90% for an IC is a composite of fault coverages for many smaller functions that make up the design. For example, a 90% fault coverage for a microprocessor is a composite fault coverage over control logic, ALU, interrupt control, I/O control, and so on. It is not uncommon for individual fault coverages to vary over a wide range. In fact, it would be unusual if fault coverages for different parts of a design were all within one or two percentage points of the composite fault coverage.

The profiler reads the master fault file and extracts results for modules identified by the user. For example, the interrupt logic in a microprocessor might be spread across several submodules grouped together under a top-level module identified as INT. The user can request fault coverage statistics for INT and for all submodules contained in INT. Alternatively, the user may request that the profiler list only the undetected faults in that section of logic.

If fault coverage for a particular module is unsatisfactory, the user can request a further breakdown. Suppose that a microprocessor contains a register bank made up of 16 registers, and that a small subset of them were used constantly during design verification, to the exclusion of all other registers. A fault coverage profile will reveal that the register bank has unacceptably low fault coverage. A further request for more details from the profiler can give additional details, showing fault coverage for each individual register. Being able to zoom in and spot those precise functions that have poor coverage is a significant productivity enhancer. Rather than blindly create test vectors and fault simulate in the hopes that fault coverage will improve,

the profiler makes it possible to explore specific areas of a design and identify those in need of improvement.

Knowing where undetected faults reside sometimes is enough to improve coverage with minimal effort. Consider the aforementioned register bank. If for some reason they are overlooked during generation of a test, the profiler can reveal that fact immediately and, once it known, all that is required is that load and store instructions be executed to test these registers. The fault coverage is then improved with negligible effort. An important side effect of this strategy is a higher quality test. It has been reported that a test in which several functions have approximately equal coverage will generally experience fewer tester escapes than another test with the same total fault coverage, but with the coverage more unevenly distributed across the modules.[14]

7.7.10 Fault Dictionaries

During fault simulation it is common for several faults to be detected by each test pattern. When testing a printed circuit board it is desirable to isolate the cause of an erroneous output to as small a group of candidate faults as is practical. Therefore, rather than stop on the first occurrence of an output error and attempt to diagnose the cause of an error, a tester may continue to apply patterns and record the pattern number for each failing test pattern. At the conclusion of the test, the list of failed patterns can be used to retrieve diagnostic data that identifies potential faults detected by each applied pattern. If one or more faults are common to all failed patterns, the common faults are high-probability candidates.

To assist in identifying the cause of an erroneous response, a fault dictionary can be used. A *fault dictionary* is a data file that defines a correspondence between faults and symptoms. It can be prepared in several ways, depending on the amount of data generated by the fault simulator.[15] If the ith fault in a circuit is denoted as F_i, then a set of binary pass–fail vectors $F_i = (f_{i1}, f_{i2}, \ldots, f_{in})$ can be created, where

$$f_{ik} = \begin{cases} 1 & \text{if } f_i \text{ is detected by test } T_k \\ 0 & \text{otherwise} \end{cases}$$

These vectors can be sorted in ascending or descending order and stored for fast retrieval during testing. During testing, if errors are detected, a pass–fail vector can be created in which position i contains a 1 if an error is detected on that pattern and a 0 if no error is detected. This vector is compared to the pass–fail vectors created from simulation output. If one, and only one, vector is found to match the pass–fail vector resulting from the test, then the fault corresponding to that pass–fail vector is a high-probability fault candidate. It is possible of course that two or more nonequivalent faults have the same pass–fail vector, in which case it is possible to distinguish between them only if they have different symptoms; that is, they fail the same test pattern numbers but produce different failing responses at the output pins.

Example The following table lists four tests and pass–fail vectors corresponding to five failing circuits, f_1 through f_5.

	T_1	T_2	T_3	T_4
F_1	0	1	0	0
F_2	1	1	0	1
F_3	0	0	1	0
F_4	0	1	0	0
F_5	1	0	0	1

■■

Faults f_2 and f_5 are both detected by test T_1. If tests T_2 and T_4 also fail, then the vector F_2 matches the pass–fail vector. If T_4 is the only additional test to fail, then F_5 is a match. Faults f_1 and f_4 have identical pass–fail vectors. The only hope for distinguishing between them during testing is to compare the actual output response to the predicted response for faulty circuits f_2 and f_4.

Because the matrices are quite sparse, it is generally more compact to simply create a list of the failing test numbers for each fault. The fault number then serves as an index into the list of failing test numbers for that fault. Then, when one or more tests fail at the tester, the fault simulator output indicates which faults are the potential cause of each test pattern failure. These faults are used to access the fault dictionary to find the fault for which the failing test numbers most closely match the actually test failures observed at the tester.

Test generation and fault simulation are based on the single fault assumption; hence the fault list for a failing test can be inaccurate. This is especially true on the first few patterns applied to a circuit since that is when gross defects are most frequently detected. However, after the first few patterns, gross defects have usually been detected and there is a growing likelihood that errors are the result of single stuck-at faults. In that case the fault data recorded by the simulator for each pattern becomes more reliable as a source of diagnostic data. Nevertheless, even without the presence of gross physical defects, unmodeled faults such as noise, crosstalk, or parametric faults produce error symptoms that are not always detectable by fault dictionaries.

7.7.11 Fault Dropping

In the past, when PCBs were made up of many discrete components, fault dictionaries were a popular means of diagnosing and repairing these PCBs. At that time the stuck-at fault model more closely approximated many of the fault mechanisms that occurred on the PCB. In addition, the number of logic elements in the circuit was much smaller, so fault dictionaries were more practical. Fault dictionaries are not as popular as they once were, because circuits have increased in size to the point where the amount of storage required for diagnostic data is simply too great. Another

problem is the fact that fault simulation of large circuits takes exorbitant amounts of CPU time. The number of faults for a typical, gate-level circuit usually runs, on average, about two and a half faults per logic gate. To simulate every fault on every pattern becomes impractical.

For PCBs, automatic test equipment can isolate faults by means of probing algorithms. In such cases, diagnostic data are not required so there is no need to continue simulating a fault after it has been detected, thus permitting it to be deleted from the fault list. This process, called *fault dropping*, can significantly speed up simulation. If full fault simulation is impractical, but diagnostic data is required, then a possible compromise between full fault simulation and fault dropping is to keep a count of the number of times that a fault has been detected. After the fault has been detected some specified number of times, it is dropped from further simulation.

The criterion for determining when to drop a fault is a function of circuit size and the number of faults detected with each pattern. The objective is to reduce simulation time while obtaining enough information to minimize the number of components that must be replaced on a board in order to restore it to proper operation. The problem is complicated by the fact that equivalent faults will always appear together if they have not been reduced to a single equivalent fault. For diagnostic purposes the amount of CPU time can sometimes be reduced if the ATPG is required to create patterns for maximum resolution rather than maximum comprehension. More test patterns are created, but fewer faults are detected by each pattern; thus fault resolution is achieved more quickly and faults are dropped sooner.

If a fault contained in a list of faults for the *n*th test pattern is the only previously undetected fault in that list, it can be dropped from further simulation. The reasoning here is that if any of the other faults actually exist in the device being tested, then during testing they will cause an output error on an earlier pattern. If the *n*th pattern is the first to fail, then the lone previously undetected fault is the likeliest fault to have occurred.

7.8 BEHAVIORAL FAULT MODELING

In previous sections we looked in detail at fault modeling. It is important to bear in mind that a fault model is exactly that, a model. As such it is an imperfect replica. Faults are modeled as SA1 and SA0 on AND gates and OR gates. However, as we saw in Section 2.13, networks of transistors do not always bear a physical resemblance to corresponding gate-level models. The purpose of the gate-level model is to limit the scope of the problem. By using logic gates, some accuracy is sacrificed, but it is possible to expedite a solution. If a problem requires too much detail it may not be solvable in reasonable time. However, if too much accuracy is sacrificed, the answer becomes meaningless. It is necessary to strike a balance.

Standard cell libraries typically contain a detailed layout describing the physical implementation of a cell, and a description of the behavior of that cell at the logic

level. A significant amount of effort goes into the design of standard cell libraries to ensure that the behavior of each member is described as accurately as possible, both with respect to logic behavior and with respect to propagation delays from input pins to output pins. However, as we previously saw, matching logic behavior to transistor-level implementation with enough accuracy to detect all physical defects is no trivial task. The task can become even more of a challenge as we look at behavioral modeling of circuits.

7.8.1 Behavioral MUX

A problem with gate-level modeling of functions is that different technologies employ different basic building blocks. The NAND gate is natural for CMOS, and the NOR gate is natural for ECL. The NAND conveniently implements a sum of products whereas the NOR more conveniently implements a product of sums. The circuits in Figure 7.9 are implemented as

$$F = (\bar{x}_1 + x_2 + \bar{x}_3 + x_4) \cdot (\bar{x}_1 + \bar{x}_2 + \bar{x}_3 + \bar{x}_4) \cdot (x_1 + \bar{x}_2 + x_3 + \bar{x}_4) \cdot (x_1 + x_2 + x_3 + x_4)$$

or

$$F = x_1 \cdot \bar{x}_2 + \bar{x}_1 \cdot x_2 + x_3 \cdot \bar{x}_4 + \bar{x}_3 \cdot x_4$$

depending on which technology is chosen to implement the function.

While behavioral models of common functions can be too abstract to permit accurate, detailed analysis of defect activity, gate-level models are also vulnerable. In fact, sometimes, ironically, behavioral descriptions can produce better tests. Consider the simple 2-to-1 multiplexer in Figure 7.14. Once again, we represent both the sum-of-products and product-of-sums versions of the circuit. The following table lists four vectors and the faults detected at the NAND circuit and at the NOR circuit.

				Faults Detected	
A	*B*	*C*	*F*	(NAND)	(NOR)
0	1	0	0	1.1, 2.1 SA1	3.1 SA0
1	0	1	0	1.2, 2.2 SA1	3.2 SA0
X	1	1	1	3.2 SA1	2.2 SA0
1	X	0	1	3.1 SA1	1.1 SA0

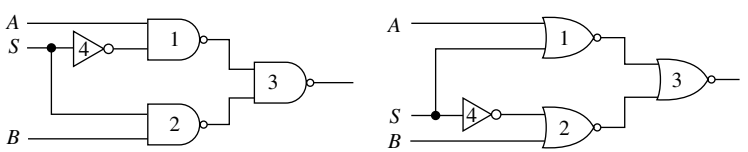


Figure 7.14 Two implementations of the 2-to-1 multiplexer.

We consider six faults in each circuit. For the NAND (NOR) circuit we consider SA1 (SA0) on each input of the three NAND (NOR) gates. All six of the faults in the NAND circuit are detected. However, only four of the six faults in the NOR circuit are detected. Input 2 of NOR gate 1 and input 1 of NOR gate 2 may or may not be detected, depending on which value is assigned to the don't cares.

An alternative view of the multiplexer as a functional entity is provided by the following Verilog equation:

$$f = (Select)\ ?\ A : B;$$

In this equation, if *Select* is 1, then $f = A$, else if $Select = 0$, $f = B$. Faults in the functional unit can be classified as control faults or data faults. The data faults are as follows:

1. Cannot propagate 0 through A.
2. Cannot propagate 1 through A.
3. Cannot propagate 0 through B.
4. Cannot propagate 1 through B.

The control faults are as follows:

5. Select A, got B.
6. Select A, got both ports, that is, A + B.
7. Select B, got A.
8. Select B, got A + B.

The eight functional faults can be detected with the following four test vectors.

A	*B*	*C*	*F*	Faults Detected
0	1	0	0	1,5,6
1	0	0	1	2
1	0	1	0	3,7,8
0	1	1	1	4

Comparing this table with the previous table suggests that the don't cares in the previous table should be set to 0. If we set them to 0 and again check the faults in the NOR gate model of the multiplexer, we find that the previously undetected faults have now been detected.

The preceding results can be generalized to any multiplexer. For an n-to-1 MUX, $2n$ tests verify that 0 or 1 can be propagated through the n ports. Selection of the wrong port is detected by using the same $2n$ vectors and putting values on other ports that are complementary to the value placed on the selected port. With the single-fault assumption it is not necessary to put opposing values on all ports.

For a 4-to-1 MUX with two select lines, S_1 and S_0, port 1 is selected by setting S_1, $S_0 = (0, 0)$. A single select line fault is likely to select either port 2 ($S1$, $S_0 = 0, 1$) or port 3 (S_1, $S_0 = 1, 0$) but not port 4 (S_1, $S_0 = 1, 1$).

Other functional entities can be similarly processed. The objective is to identify invariant properties common to all or most physical realizations. Then, effective tests can be created without detailed structural descriptions. There is the added advantage that test pattern generation can be started before the design has been completed. Basic functional entities include:

Elementary gates: AND, OR Invert, simple combinations
Latches, flip-flops: JK, D, T
Multiplexers
Encoders and decoders
Comparators
Parity checkers
Registers
ALUs: logic, arithmetic—fixed point, binary coded decimal (BCD), floating point
Memory arrays
State machine

In the final analysis, fault models are used to evaluate the effectiveness of test vectors for detecting physical defects in logic circuits. To that end, the modeling of faults for functional primitives should reflect the types of physical defects that are likely to occur and their effect on functional behavior. For example, a binary counter with parallel load capability must be able to perform a parallel load, it must be able to advance the count to the next higher binary stage, and it must be resettable. A physical defect that alters any of these functional capabilities must be modeled in terms of its effect on the function.

The fault model must reflect device behavior when the fault is present, because it is only by simulating the behavior of the faulted circuit and observing the consequences of that behavior at an output pin that detection can be claimed for the fault. For example, if the output of the ith flip-flop in a counter is SA1, then the counter begins counting with an initial value of 2^i rather than 0 following a reset. In normal operation, when counting up, bit position i resets to zero when bit position $i + 1$ switches to 1. To simulate faulted operation it must be forced to remain at 1.

7.8.2 Algorithmic Test Development

When performing fault-directed testing, an ATPG, or a test engineer, selects a particular fault and generates a test for that fault. However, for memories, fault-directed testing is not used. Because memories have a regular structure, it is possible to apply very concise algorithmic test programs that test them more thoroughly with less effort on the part of the test engineer. These algorithmic programs test not only stuck-at faults, but many other kinds of defects as well (cf. Chapter 10).

Other functions are amenable to algorithmic test patterns. These tests fall into the category of *black box* tests; that is, these are tests developed without any visibility into the structure of the device being tested. We begin with the n-wide bus. It could be an address or data bus connected to memory, or any other data path requiring two or more wires to carry data into or out of some functional unit. Assume that $2^{i-1} < n \leq 2^i$, for i an integer and $i > 0$. Then construct an $i \times n$ matrix by means of the following code:

```
i = 5;  // no. rows in matrix == log2(bus width)
n = power(2,i); // n = 2**i
for(k = 1; k <= i; k++) {   // row k of matrix
  limit = power(2,i-k); // limit = 2**(i-k)
  for(m = 0; m < n/limit; m++)
    for (p = 0; p < limit; p++) {
      index = limit * m + p; // create p zeros
      row[index] = m % 2; // followed by p ones
      fprintf(stderr, "%c", row[index]+'0');
    }
  fprintf(stderr,"\n");
}
```

The matrix created by this C program, when $i = 5$, is as follows (the last line was added manually):

```
00000000000000001111111111111111
00000000111111110000000011111111
00001111000011110000111100001111
00110011001100110011001100110011
01010101010101010101010101010101
1XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX0
```

For an n-bit bus, this matrix checks that each wire can propagate both 0 and 1. Note that the rightmost column did not receive a 0, and the leftmost column did not receive a 1, hence the need to manually add an $(i + 1)$st row. It is worth noting that this matrix also checks for "stuck-to-neighbor" faults. Pick any two columns j and k, then the values in columns j and k will differ in one of the vectors. That follows from the fact that the columns generate every possible combination from 0 to $2^i - 1$. Whether two nets with different values assume the value 0 or 1 in the presence of a bridging fault depends on the technology. An interesting observation: Whenever the number of bus bits doubles, a single additional vector is required.[16]

Now consider the possible faults that could occur in the ith stage of an n-bit binary counter. The output of the ith stage could be SA0 or SA1. If the counter has parallel load capability, these faults can be revealed by loading all 0s and all 1s. If the counter does not have a load capability, a clear operation can force the counter to all 0s and

reveal an SA1 at the output of the *i*th stage. More challenging are the interstage connections. When the current value in the counter is $2^i - 1$, an active clock edge causes the *i*th stage to switch from 0 to 1 and all lower stages switch from 1 to 0 if the counter increments correctly. If all stages up to the *i*th stage are 1s, and the *i*th stage is 0, this 0 blocks a carry from propagating to higher stages. If the counter is decrementing, a borrow propagates through 0s until it reaches a stage whose value is 1.

If a carry into stage $i + 1$ is SA1, then clearing and clocking the counter will cause the $(i + 1)$th stage to switch to 1. On the other hand, if the carry logic is SA0, the sequence generated by the code below will reveal the fault. The following Verilog code illustrates these operations (keywords are highlighted to improve readability). If a commercial Verilog simulator is not available, the Icarus Verilog simulator (www.icarus.com) can be used to simulate the example. The output is written into a file called response.fil.

```
module b16ctr(ctrout,din,clk); // behavioral 16-bit
                               // counter
output [15:0] ctrout;
input [19:0] din;
input clk;
wire loadall  = din[3], incrcntr = din[2];
wire decrcntr = din[1], reset    = din[0];
reg [15:0] ctrout;
wire load = loadall & reset;
always @(posedge clk) begin
    if(load == 0)
      ctrout <= din[19:4];
    else if((incrcntr == 1) | (decrcntr == 1))
      ctrout <= (decrcntr == 1) ? ctrout - 1 : ctrout + 1;
  end
endmodule
module testbench;
reg [19:0] din;  // din[3:0] = (load, incr, decr, reset)
reg clk;
wire [15:0] ctrout;
integer i, response;
b16ctr X1 (ctrout, din, clk);
initial begin
    response = $fopen("response.fil");
    #1 clk = 1'b1;
   end
always begin
    #24 clk = ~clk;
    #1  if(clk == 1)
```

```
    $fdisplay(response, $time,"  %b %b %b %b",
      clk, din[19:4], din[3:0], ctrout);
  end
always begin
    #1
    $fdisplay(response,"// check propagate circuits");
      vec_gen(1'b1, 4'b1101, 20'h0);
    $fdisplay(response,"// check borrow circuits");
      vec_gen(1'b0, 4'b1011, 20'hFFFF0);
    $fdisplay(response,"// check propagate inhibit");
      vec_gen(1'b1, 4'b1101, 20'hFFFE0);
    $fdisplay(response,"// check borrow inhibit");
      vec_gen(1'b0, 4'b1011, 20'h00010);
    $fclose(response);
    $finish;
  end
task vec_gen;
input shift_in;
input [3:0] control_bits;
input [19:0] all_din;
  begin
    din[19:0] = all_din;
    for(i = 0; i < 16; i = i+1) begin
      #50; din      = {din[18:4], shift_in, 4'b0001};
      #50; din[3:0] = control_bits;
    end
  end
endtask
endmodule
```

We next consider a fixed-point ALU. The following Verilog RTL code describes the 74181, a 4-bit ALU slice that was once commonly used as a discrete component on printed circuit boards and which has since served as a template for ALU macrocells for many component libraries (cf. Figure 7.23).

```
module xy (X,Y,S3,S2,S1,S0,A,B);
input S3, S2, S1, S0;
input  A, B;
output X, Y;
wire X = !(A & (S3 &  B | S2 & !B));
wire Y = !(A |  S0 &  B | S1 & !B);
endmodule
```

```
module sn74181(F3,F2,F1,F0,A_EQ_B,P,CN4,G,S3,S2,S1,S0,
     A3,A2,A1,A0,B3,B2,B1,B0,M,CN);
output F3, F2, F1, F0;
output  A_EQ_B;
output  P, CN4, G;
input  S3, S2, S1, S0;
input  A3, A2, A1, A0;
input  B3, B2, B1, B0;
input  M, CN;
wire X3, X2, X1, X0, Y3, Y2, Y1, Y0;
wire CN4 = !G | X0 & X1 & X2 & X3 & CN;
wire P   = !(X0 & X1 & X2 & X3);
wire G   = !(Y0 & X1 & X2 & X3 | Y1 & X2 & X3 | Y2 & X3
                | Y3);
wire F3 = X3 ^ Y3 ^ (M | !(CN & X0 & X1 & X2 | Y0 & X1 &
                 X2 | Y1 & X2 | Y2));
wire F2 = X2 ^ Y2 ^ (M | !(CN & X0 & X1        | Y0 & X1
                 | Y1));
wire F1 = X1 ^ Y1 ^ (M | !(CN & X0             | Y0));
wire F0 = X0 ^ Y0 ^ (M | !(CN));
wire  A_EQ_B = F3 & F2 & F1 & F0;
xy U3 (X3,Y3,S3,S2,S1,S0,A3,B3);
xy U2 (X2,Y2,S3,S2,S1,S0,A2,B2);
xy U1 (X1,Y1,S3,S2,S1,S0,A1,B1);
xy U0 (X0,Y0,S3,S2,S1,S0,A0,B0);
endmodule
```

When $S = \{1,0,0,1\}$ and $M = 0$, the 74181 performs an add operation, $F = A + B + CN$. For the add operation, $X_i = !(A_i \,\&\, B_i)$; and $Y_i = !(A_i \mid B_i)$. With these values a typical term F_i becomes

$$F_i = !(Y_i \,\hat{}\, X_i \,\hat{}\, (Y_{i-1} \mid X_{i-1} \,\&\, Y_{i-1} X_{i-1} \,\&\, X_{i-2} \,\&\, Y_{i-2} \mid ..... \mid X_{i-1} \,\&\, ... \,\&\, X_0 \,\&\, C_N));$$

An algorithmic test will be described next that controls the X_i and Y_i signals by means of the A_i and B_i signals. A significant part, but not all, of the circuit elements can be tested using the add operation. For example, when performing the add operation, the combination X_i, $Y_i = \{0,1\}$ cannot be achieved. That combination can be obtained by selecting logic operations for the op-code S. The following Verilog code implements the algorithm for an 8-bit data path:

```
module testbench;
reg [8:0] A, B, WALK;
reg Cin;
```

```
wire [7:0] F;
integer i, j, response;
alu  X1 (A[8:1], B[8:1], Cin, F);
initial
  response = $fopen("response.fil");
always begin
    for(j = 0; j < 9; j = j+1) begin
      Cin = (j == 0) ? 1 : 0;
      WALK = 9'b1 << j;
      for(i = j; i <= 8; i = i+1) begin
        A[8:0] = 9'b0 ^ WALK;
        B[8:0] = 9'h1FF ^ WALK << i-j+1;
        #10 $fdisplay(response, $time," %b %b %b %b",
          A[8:1], B[8:1], Cin, WALK);
      end
      $fdisplay(response, "");
      end
    $fclose(response);
    $finish;
  end
endmodule
```

As mentioned before, any Verilog simulator will run the code and write the results into the file response.fil. For an n-wide ALU, the algorithm generates $n \cdot (n + 1)/2$ vectors. This test walks a 1 across the A port. That 1 is added to the argument at the B port to create generate and propagate signals. The A and B arguments can be reversed and the test applied again. After this algorithmic test has been run, a small number of logic operations can be performed to detect the remaining undetected faults.

When an algorithmic test exists for a particular function, it can be used for design verification as well as for manufacturing test. The Verilog code needed to drive the circuit through a series of state transitions that deliver the ALU operands to the ALU ports can be added to the Verilog code to make a complete test.

Although the logic designer may only be concerned with confirming that the function is correctly wired to the rest of the circuit, a comprehensive, prepackaged algorithmic test that detects all faults will serve two purposes: It will verify that all inputs are connected correctly to the rest of the circuit, and it will serve as an effective manufacturing test. Such tests are, like memory tests, often easy to program concisely. Note that an algorithmic test is not necessarily the smallest test, in terms of vector count. For another view, directed toward determining the smallest set of vectors, see Section 7.9.5.

7.8.3 Behavioral Fault Simulation

The advent of RTL logic design and the resulting reliance on logic synthesis has had a major impact on design styles and productivity. By expressing a design at a

higher level of abstraction, the designer can focus on circuit behavior until the model responds correctly. However, from the standpoint of developing and evaluating test programs, RTL design introduces its own problems. We discussed the implications of granularity in Section 3.4. While it would be desirable to fault simulate at the RTL level, the level of granularity is so coarse that results may be totally meaningless. The fault coverage number, which is a metric whose purpose is to quantify the goodness or thoroughness of a test, may be deceptively optimistic. As an example, it was pointed out in Section 7.5.6 that fault coverage of manufacturing faults is often far more optimistic than fault coverage of field faults for the same circuit.

Fault simulation at the RTL level may be desirable in order to propagate faults through behavioral models that do not have structural counterparts, or it may be desirable in order to evaluate the quality of a test. If the purpose is to propagate faults through behavioral elements that do not have gate-level counterparts, a preferable alternative may be to synthesize the circuit into a gate-level model. If that is not practical, then fault simulation at the RTL level can be accomplished in a concurrent fault simulator by processing the behavioral module(s) in the same way that the built-in primitives are processed; that is, when a fault effect arrives at one or more inputs to the behavioral module, a pointer to that module is placed on the time wheel. At the appropriate time the module is evaluated and the fault effects are further propagated (cf. Section 3.7).

It may be desirable to fault simulate RTL modules in order to get a first-order estimate of fault coverage. This can be helpful in spotting testability issues before a design is synthesized. Test-resistant logic can then be redesigned before synthesis is performed. In such cases, physical defects must be modeled realistically, so as to satisfy the criteria of Section 3.4 and permit faults to be simulated accurately and quickly.

Fault insertion in functional models can be accomplished in a variety of ways. The simplest way, for individual faults, is to introduce a fault variable v into an expression such that the expression evaluates correctly if $v = 0$, indicating that the fault is not present, and incorrectly when $v = 1$, indicating that the fault is present. Notationally, this can be expressed as

$$F = \bar{v} \cdot f_g + v \cdot f_f$$

where f_g denotes response for the good circuit and f_g denotes response for the faulted circuit. If a function has many possible faults, it usually requires less CPU time if, whenever possible, a single multivalued fault variable is used to specify either the unfaulted function or one of n faulted models. Then, the fault variable is set before the function is evaluated. Upon entering the function, the fault variable is evaluated once. For a 2-to-1 mux, the following *case* statement determines whether the fault-free code or code corresponding to a particular fault is executed.

```
reg [15:0] fault_num;
case (fault_num)
  16'd0:
```

```
  16'd1: A = 1;
  16'd2: A = 0;
  16'd3: B = 1;
  16'd4: B = 0;
  16'd5: A = B;
  16'd6: A = A | B;
  16'd7: B = A;
  16'd8: B = A | B;
endcase
case (S)
  0: F = A;
  1: F = B;
  X: if (A == B);
       F = A;
     else
       F = X;
endcase
```

The fault number *fault_num* determines which case statement is executed. Case 0 corresponds to the fault-free circuit. After a fault is inserted, the second case statement executes the simulation code. If the control signal is indeterminate, but the inputs match, the output is set equal to the inputs; otherwise it is set equal to X. If A and B are m-bit wide ports, then a more detailed bit-by-bit comparison is necessary.

What happens when the case statement is incomplete? A simple solution is to ignore the effects of faults for which behavior is undefined. In a case statement that decodes op-codes, the default may be to take no action for op-codes that are unrecognized. Such a fault then becomes undetectable, unless it can propagate to an output by way of some other signal path. If the purpose of the case statement is to decode op-codes, then a possible solution is to load the model's Instruction Register with Xs. The fault may then eventually become a probable detect.

For more complex functions, such as a CPUs, additional complications arise. Simulation during design verification may be performed at far too high a level of abstraction to permit meaningful fault analysis. In such a case it may be possible to break a behavioral module into several smaller submodules and apply SA0 and SA1 faults on each input and output pin of each of these submodules. This provides greater granularity and may help to identify paths that fail to get exercised when writing verification suites.

A more meaningful fault estimate may be obtained by performing operations on arguments at a lower level of abstraction. For example, in an ALU, a fault simulation result may be meaningless if the operation $F = A + B$ is performed at the behavioral level, particularly when one or both arguments have indeterminate values. But, if simulation is performed by adding bits iteratively from lowest to highest bit position, including the propagate and generate bits P_i and G_i (see Section 7.8.2), then

fault simulation results may be meaningful, even when some of the bit positions are indeterminate. The sum begins at the carry-in and proceeds from low-order to high-order bit position. If an input bit position is indeterminate in one vector, but the other input and the carry-in are both 0s, the indeterminate value is blocked from propagating to the next higher position. The iterative method lends itself to any argument size since the number of iterations can be an argument in a loop control statement.

The iterative approach also permits simulation of faults internal to the ALU. However, all the P_i and G_i must first be computed, based on the values A_i and B_i. Then, as with the MUX previously described, an individual A_i, B_i, P_i, or G_i is faulted, based on the fault number. The ALU result is then computed for either the fault-free or some faulty circuit. The sum at position i is computed using A_i, B_i and a carry C_i into position i where

$$C_i = G_{i-1} + P_{i-1} \cdot C_{i-1}$$

The G_i and P_i can be computed once. Then, individual parameters can be faulted and the effects computed in a loop until all fault effects have been analyzed.

7.8.4 Toggle Coverage

It was pointed out in Section 7.5.2 that checkpoint faults, barring redundancies in the netlist, uniquely correlated to 2-tuples of type <signal path, logic value>. High coverage of these faults, using design verification vectors, usually indicates a thorough design verification suite—that is, one that checks most, if not all, of the obscure corners of a design. This raises the following question: If a test suite thoroughly exercises an RTL design, does it also give good fault coverage? Expressed another way, high coverage of an RTL design is necessary if a verification suite is to provide high fault coverage, but is it sufficient? Before addressing that question, we address the following question: "How do we determine whether a test suite provides thorough design verification coverage?"

One method that has been used for many years is *toggle coverage*. This operation keeps track of the number of times each net in a circuit switches from 0 to 1 and from 1 to 0. For a bus driven by two or more tri-state drivers, the operation may count the number of transitions to and from the high-impedance state as well. Toggle coverage is performed during gate-level simulation, and it is quite easy to compute the count at that level of detail.

Toggle coverage can be used to advantage to determine where "hot spots" exist on an IC. In CMOS ICs power consumption is proportional to switching activity. It is possible that total power consumption in an IC is well under some predetermined upper limit. However, it may be the case that a large amount of power consumption occurs in a relatively small area of a chip where a disproportionately large amount of switching activity takes place. By performing toggle counts during gate-level logic simulation and linking switching activity to X-Y coordinates on the die, it is

possible to identify areas of the die where this concentrated switching activity causes local hot spots that lead to premature failure of the IC. If hot spots are detected, the logic can be rearranged on the die and resimulated with the toggle count option. It is important to note that gate-level simulation must be performed with nominal or back-annotated delays. A unit or zero-delay simulation, and particularly a rank-ordered simulation, will not accurately reflect the number of times each logic element switches in a given time frame.

For design verification, the fact that toggle count is performed at the gate level presents a problem. Since so much of IC design activity is at the RTL level, a gate-level toggle count is performed after a design has been synthesized. If toggle count reveals that verification is inadequate in some areas of the design, two problems exist. First, a synthesized design is usually difficult or impossible to trace. Arbitrary name assignments during synthesis often bear little or no correspondence to the original RTL. The larger the module, the more difficult it is to relate the gate level to the original RTL. A second problem, if design flaws are uncovered as a result of additional test suites written to improve toggle count, is that the synthesis process must be repeated. It is much preferred to identify errors in the design before it is synthesized.

7.8.5 Code Coverage

An alternative to toggle count is *code coverage*. It is measured during RTL simulation. Code coverage has been used for many years by software developers to measure thoroughness of test suites written for programs expressed in high-level languages (HLLs). For HDLs it not only can point to areas of a design where coverage is low, but also can point to areas where coverage is adequate and can thus save the designer some time. The most obvious metric is *block coverage*. It. identifies lines of code that were executed and lines that were overlooked during creation of test suites. Coverage reports can be generated on a module basis, with results identifying (a) the percentage of lines of code that were executed in each module and (b) the specific lines of code that failed to get exercised. By knowing the percentage of lines of code in each module that were covered, the user can target modules with the lowest coverage results and write tests to exercise the unverified code in those modules.

Another form of code coverage is *expression coverage*. In this mode, individual expressions are evaluated in greater detail. Consider the following expression:

$$Y = A \ \& \ B \mid C \ \& \ D;$$

Any set of values would exercise the equation, $(A,B,C,D) = (0,0,0,0)$ is one such set of values. If the only goal was to confirm that the expression had been executed, those values would satisfy the requirement. However, if this equation controlled the operation of some major function, very little information is gained from the values just cited. If we were interested in an event corresponding to variable A, we might want variable B to be a 1, in order to verify that A is able to block the event controlled by Y, or we might want A to be 1 and B to be 1, in order to verify that A is able to trigger the event controlled by Y. Furthermore, if both C and D were always 1, then any

values assigned to *A* and *B* would be blocked from having an effect downstream in the logic.

A more meaningful assessment of the equation provides feedback indicating which of the four variables controlled the outcome of the expression during simulation. Interestingly, this is precisely what fault simulation does. Expression coverage at the RTL level in a code coverage tool accomplishes something very similar to what fault simulation accomplishes at the gate level. The major difference is that code coverage only measures controllability while fault simulation measures controllability and observability; that is, a fault effect must be driven to an observable output.

A third code coverage metric is *path coverage*. In Verilog it measures the thoroughness of coverage for all possible paths through "initial" and "always" blocks, as well as within each "forever," "while," "repeat," and "for loop" construct. Fixed integers or variables used to specify the number of iterations through a loop can be checked to determine whether the full range of values was exercised. Paths through successive conditional blocks can be checked. So, if there are two successive *if...else* expressions, there are two paths through the first expression and two paths through the second, but there are four distinct paths through the two constructs taken jointly. There may be circumstances when it is desirable to verify all four paths through the code. Other forms of coverage can be evaluated using code coverage. Case statements representing state machines can be evaluated to insure that all states have been visited and that all arcs have been traversed. A case statement may represent a multiplexer, and it may be desirable to verify that all paths through the multiplexer have been exercised.

How effective is code coverage? A study was performed to compare the results of code coverage versus fault simulation, using the same test vector sequences to evaluate both operations.[17] An initial set of test vectors was captured from a design verification testbench where they were used to check out an RTL model. These vectors were reapplied to the RTL model after it had been instrumented for code coverage. The instrumentation process consists of compiling the RTL design code and embedding PLI (programming language interface) calls during the compilation. The calls kept track of which lines of code were evaluated, and they also kept track of what values appeared on the variables in those lines of code.

The same vectors were fault simulated against a gate-level model of the circuit. The results of these two operations are illustrated in Figure 7.15. The fault coverage profiles, both code coverage and fault simulation coverage, were plotted for several levels of hierarchy. The leftmost column indicated coverage for the entire design. The next few columns indicate coverage for each of the top-level submodules. Eventually, continuing down the hierarchy in this fashion, coverage at the far right is provided for the smallest modules. The dotted line indicates RTL code coverage, and the dashed line indicates fault coverage. The coverages for this particular circuit track rather well for the larger modules; it is only at the extreme right, representing modules that consist of perhaps four to eight lines of RTL code, that the correspondence breaks down.

After examining the results and identifying where the fault coverage was unacceptably low, additional test vectors were written, specifically targeting low coverage areas of the chip. These brought total coverage up to 92.35%. The two sets of

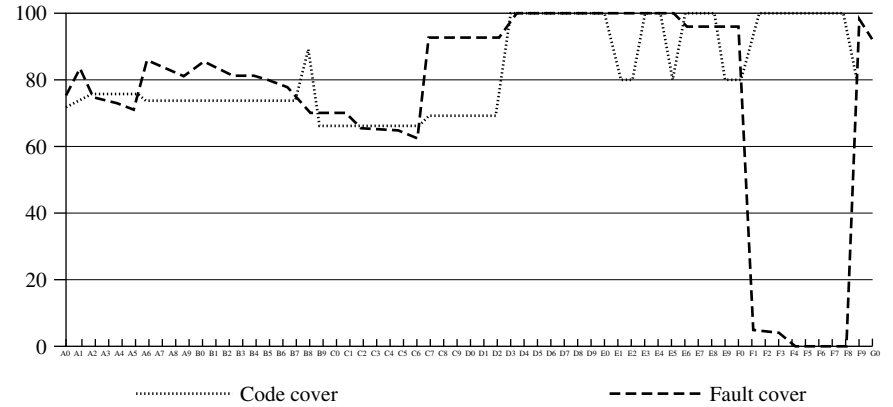

100
80
60
40
20
0
A0 A1 A2 A3 A4 A5 A6 A7 A8 A9 B0 B1 B2 B3 B4 B5 B6 B7 B8 B9 C0 C1 C2 C3 C4 C5 C6 C7 C8 C9 D0 D1 D2 D3 D4 D5 D6 D7 D8 D9 E0 E1 E2 E3 E4 E5 E6 E7 E8 E9 F0 F1 F2 F3 F4 F5 F6 F7 F8 F9 G0
Code cover
Fault cover

Figure 7.15 Fault coverage versus code coverage (80.45%).

vectors were then resimulated against the instrumented RTL model, and the results again were plotted. The correspondence between code coverage and fault coverage improved as fault coverage increased to 92.35%. This is seen in Figure 7.16.

It is interesting to note that for some modules, code coverage was higher than fault coverage, while for other modules fault coverage was higher than code coverage. One should be careful not to read too much into a single investigation. A problem with using code coverage vectors for manufacturing test is that designers are not obligated to propagate results all the way to outputs. A designer may verify that it is possible to load a register, or traverse a state machine, and stop at that point. Furthermore, the designer may load a register directly via the testbench, rather than apply signals at the inputs and propagate them through internal logic in order to load a register. This discussion can be summed up with the observation that high code coverage is a necessary, but not sufficient, condition for high-fault coverage.

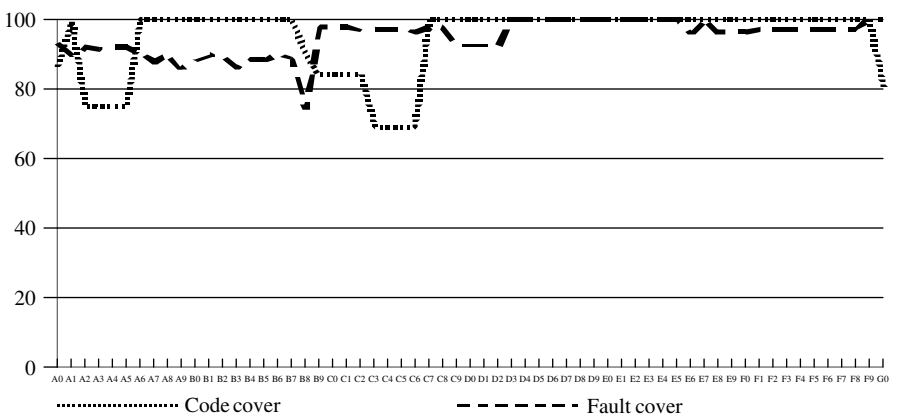

100
80
60
40
20
0
A0 A1 A2 A3 A4 A5 A6 A7 A8 A9 B0 B1 B2 B3 B4 B5 B6 B7 B8 B9 C0 C1 C2 C3 C4 C5 C6 C7 C8 C9 D0 D1 D2 D3 D4 D5 D6 D7 D8 D9 E0 E1 E2 E3 E4 E5 E6 E7 E8 E9 F0 F1 F2 F3 F4 F5 F6 F7 F8 F9 G0
Code cover
Fault cover

Figure 7.16 Fault coverage versus code coverage (92.35%).

7.9 THE TEST PATTERN GENERATOR

In Chapters 4 and 5 we examined in detail the algorithms currently used in ATPGs. In the next chapter we will examine DFT methods that have evolved to make ATPG useful as a tool for creating effective test programs. In this section we examine some methods that have been developed to either enhance the capabilities of ATPG or make it more flexible, as well as make it easier for users to tailor it to specific needs.

7.9.1 Trapped Faults

When logic signals are clocked through a sequential digital circuit, error signals produced by the faults frequently are clocked into storage elements, including latches and flip-flops. These faults are referred to as *trapped faults*. If the flip-flop clock is gated, or if the flip-flop has a hold mode, permitting it to hold existing contents or clock in new data under control of a select line, then it is possible that fault effects may remain in the flip-flop for many clock cycles. Oftentimes these trapped faults occur in registers that are remarkably easy to control and observe. For example, general-purpose registers in a microprocessor are controlled via Load and Store instructions. If a particular register contains one or more trapped faults, these trapped faults can be driven to the output bus and thus detected, simply by inserting the appropriate Store instruction.

It is a simple matter for a fault simulator to be instrumented with code to monitor the registers and identify those that contain trapped faults at any given time during fault simulation. The simulator can count the number of fault effects, if any, that become trapped in each storage element. This information can be used to prioritize the storage devices according to how many fault effects are trapped in each device. The volume of data is usually intractable during the initial stages of fault simulation, but the strategy can become valuable during the latter stages of simulation.

A comprehensive strategy employing this capability can be deployed as follows:

1. Fault simulate and update the master fault file.
2. Read in the undetected faults.
3. Resimulate the undetected faults with the trapped faults feature turned on.

The first and second steps are normal fault simulation steps. However, the third step involves rerunning fault simulation with the undetected faults and the test vectors that were previously run. No additional faults will be detected with these vectors. However, by identifying faults that become trapped in storage devices, it becomes possible to alter the test program in order to flush out some of these trapped faults. The user may be given the ability to specify registers that the ATPG should monitor. For example, general-purpose registers in a microprocessor can be directly read out with the Store instruction, so they would be candidates for monitoring. Either the vectors that are being simulated can be altered to enhance the fault coverage, or an altered version can be attached to the end of the existing vectors in order to improve the coverage.

7.9.2 SOFTG

A wealth of data about fault effects exists in the data structures of a concurrent fault simulator. Rather than pick a fault at random from a master fault list, the ATPG can target one of these trapped faults. The simulator oriented fault test generator (SOFTG) does exactly that. It inspects the results of simulation to determine if any faults are trapped in a flip-flop that is close to a primary output.[18] If it finds a trapped fault that appears to be easy to propagate, that fault is selected as the next candidate by the ATPG. Since the ATPG uses the current state of the circuit, it does not need to create an initializing sequence; rather, it only needs to create a propagation sequence.

In a typical implementation of this concept the ATPG creates a sequence of vectors and passes these on to the fault simulator, which accepts the vectors and resumes fault simulation from that point where it previously left off. Initially, during the first few vectors, there is no previous state and circuit state is indeterminate, so an initializing sequence is passed to the fault simulator. After fault simulating a sequence of vectors passed to it by the ATPG, the fault simulator turns control over to an executive routine that examines the circuit state and locates trapped faults, as indicated by the fault effects.

The executive routine then determines if the ATPG should propagate a trapped fault or target a new fault from the master fault list. A number of criteria must be considered when selecting a candidate fault. A register may have many trapped faults linked to it, or there may be a register close to an output that has several undetected faults trapped in it. It could very well be the case that faults in a register are blocked by an enable signal on tri-state buffers that control access to a bus connected to output pins. Enabling the tri-state buffers may be a very simple operation.

Some trapped faults may propagate in response to a clock edge. However, some faults may be *dead-end faults*. Figure 7.17 illustrates a situation where a select line is controlled by flip-flop *S*. If flip-flop *A* is selected and flip-flop *B* has trapped faults that we wish to propagate to the output, then it would seem to be a simple operation to select *B* and cause the trapped faults to propagate to *F*. However, in the process of setting up the *Select* line it is possible that the entire history of flip-flop *B* changes. A new value is clocked in, and all of the trapped faults disappear, to be replaced by an entirely new set of linked fault effects (or perhaps, none). In that case, any effort to propagate trapped faults will be in vain. This can be detected by the fault simulator and, when it happens, the fault simulator should be equipped with a roll-back feature permitting it to delete the added vectors, unless they detect other, untargeted, faults.

7.9.3 The Imply Operation

In his original article on the D-algorithm,[19] Roth propagated sensitized signals on one or more test paths all the way to the outputs before performing justification. In a subsequent paper,[20] Roth described a modified D-algorithm, called DALG-II, in which the full implication of every assignment is carried out at every step of the propagation or justification phase. In general, an implication exists if, as a result of

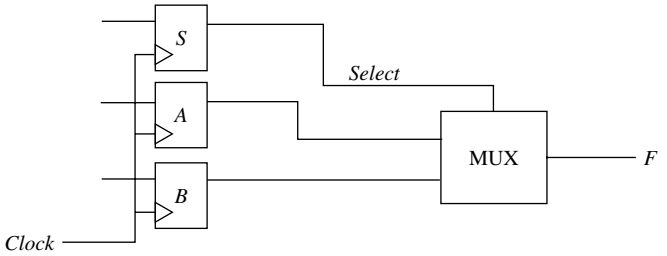


Figure 7.17 Dead-end fault.

existing assignments on the inputs and output(s) of a primitive, only one entry in the cover exists that does not conflict with existing assignments. If no entry exists, then a conflict has occurred.

Example In Figure 7.18 we want to derive a test for an SA0 on the upper input of gate *J*. We start by assigning the PDCF (1, 0) to the inputs. The 0 on the lower input implies 1s on *D* and *E*. A 1 on the output of gate *I* and a 1 on the input from *D* implies a 0 on the output of *G*. That implies 1s on inputs *B* and *C*. Finally, a D propagates through *J*. That requires a 1 on the upper input to *K*. Input *B* was previously assigned a 1, so a 0 is implied on input *A* and the test is complete. ■■

When decisions are encountered, they can frequently be postponed. Gate-level test pattern generation is one endeavor where it is desirable to put off making decisions whenever possible. We avoided a decision in the example just described by starting with the lower input to gate *J*. If the upper input had been selected first for processing, then a decision would be required as to which input to gate *I* would be assigned a 0. That could have caused a 0 to be assigned to input *D*, resulting in a conflict. By postponing the decision, it was ultimately avoided. The general rule is to avoid making decisions as long as any alternate activity can be performed. When decisions are made, it is necessary to record enough information so that if a decision leads to a conflict, it is possible to restore the circuit to the state that existed when the decision was made. This permits an alternate decision to be made and evaluated.

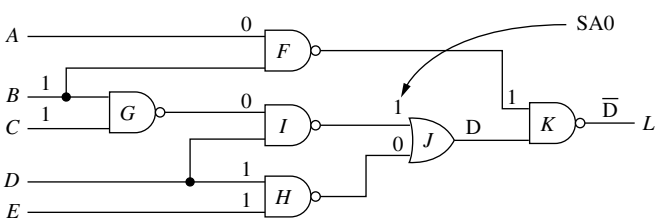


Figure 7.18 The implication operation.

7.9.4 Comprehension Versus Resolution

When creating test stimuli for digital circuits it is possible to bias the algorithm for either maximum or minimum fault detection with each pattern. If it is only necessary to determine whether an IC is good or bad, and there is no requirement to diagnose the cause of a failure, then we may want to make that determination with a minimum number of vectors; that is, we want maximum fault coverage or comprehension with each test vector. Minimizing the number of test vectors will reduce the amount of CPU time required for fault simulation. Furthermore, it can reduce the amount of storage space required to store stimulus and response data at the test station. On the other hand, when testing a printed circuit board that may contain up to 200 IC packages, it is desirable to locate a failed IC so that the board can be repaired. This can often be done more easily if fewer failures are detected by each test pattern.

The algorithm can be biased by applying propagating or nonpropagating input values to primitives during the justification phase. This is illustrated in the circuit of Figure 7.19. When testing the output of gate 10 SA0, we may select (0, 0) for the inputs or we may select either of (0, 1) or (1, 0). If we select (0, 0), then no fault on preceding logic will propagate through the NAND gate and the only fault detected is the output of gate 10 SA0. If (1, 0) or (0, 1) is selected, then other faults can propagate through gate 8 or 9 to the output.

The concept of deadening, or desensitizing, propagation paths in order to increase resolution can be enhanced by initially selecting faults at or near primary outputs and desensitizing signal paths at every opportunity. Maximizing comprehension when using the D-algorithm may be achieved in combinational circuits by initially selecting faults at or near the inputs and selecting propagating values whenever possible. It can also be achieved by using dynamic compaction, as explained in the next section, or the subscripted D-algorithm (Section 4.5).

Another feature proposed by Roth for DALG-II is the "fast plunge." Frequently, at fanout points, the next gate selected for propagation is the lowest numbered gate in the fanout list. In the circuit of Figure 7.19, a D on input 1 would be propagated through gate 5. However, the fast plunge selects the highest numbered gate, in this instance gate 8, and propagates through it rather than through gate 5. Since rank ordering assigns higher numbers to gates furthest from primary inputs, the algorithm will often get to an output in a smaller number of steps, and with fewer gate assignments requiring justification. Another motive for selecting a gate other than the lowest numbered gate in the fanout list is that, because of reconvergent fanout, it may be more difficult to propagate through a lower numbered element.

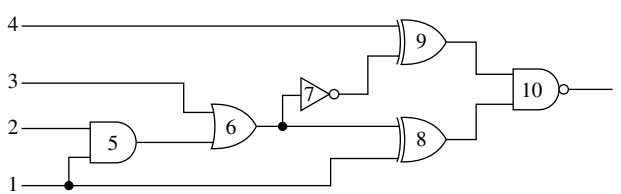


Figure 7.19 Extending a sensitized path.

7.9.5 Probable Detected Faults

We looked at probable detects in some detail in Section 5.2.1. It was pointed out that some faults in sequential circuits make it impossible to drive the circuit into a known state. When the fault-free circuit is able to enter a known state, it is possible to predict the correct value at an observable output. However, because the response of the faulty circuit is indeterminate, it might respond with the same value as the fault-free circuit, or it might produce a value that differs from the fault-free circuit. We can tell if a flip-flop is responding correctly by observing whether or not it is capable of propagating both logic values.

Consider the circuit in Figure 7.20. If the CLK input is SA1, the output of the flip-flop is indeterminate. However, in a properly working flip-flop the output follows the input when an active edge is applied to the clock. Hence, we can require that it be marked as a 1/*x* detect if the fault-free flip-flop has a 1 on its output, and if that value is propagated to the output. If the fault-free flip-flop has a 0 on its output and if the output of the flip-flop is detected, then we can mark it as a 0/*x* detect. If both 1/*x* and 0/*x* detects occur, then the stuck-at fault on the CLK input can be marked as detected.

7.9.6 Test Pattern Compaction

Quite often a test for a given fault requires assigning values to relatively few of the primary inputs. If there are several patterns with a small number of input values assigned, then pairs of these test patterns can frequently be merged, provided that none of the input positions conflict. The general rule for merging is:

If one vector has a 1 in position *i* and the other vector has a 0 in position *i*, they cannot be merged.

If one vector has $e \in \{0,1,X\}$ in position *i* and the other has X, then position *i* is assigned the value *e*.

This process is called *static compaction*. Sequences of vectors can also be merged. When self-initializing sequences of test patterns are created for sequential circuits, as is done when employing the iterative test generator, an entire sequence can be placed immediately following another sequence. However, the number of test patterns can sometimes be significantly reduced by merging sequences.

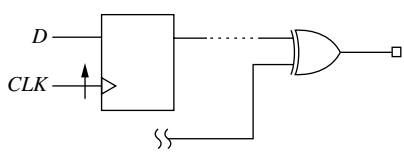


Figure 7.20 Counting probable detects.

Example We will attempt to merge the following two sequences of patterns.

	(1)							(2)						
1:	1	X	0	0	1	1		1:	X	1	0	1	X	1
2:	0	0	X	0	1	0		2:	0	0	X	1	0	X
3:	1	1	1	0	X	0		3:	0	0	X	X	1	1
4:	0	1	0	X	1	X		4:	1	X	X	1	0	0
5:	X	X	1	1	X	1								

We start with the first pattern of the second sequence and compare it with the last pattern of the first sequence. There is a conflict in the third bit position. We then compare it to the fourth pattern of the first sequence. This time there is no conflict. However, we cannot simply merge the patterns because the sequences are chronologically dependent. All four patterns of the second sequence must be applied in strict sequence. Therefore, it is necessary to compare the second pattern of the second sequence with the last pattern of the first sequence. If they conflict, the sequences cannot be merged. In this case there is no conflict so the two sequences can be merged by combining the last two patterns of the first sequence with the first two of the second sequence. This produces

1:	1	X	0	0	1	1
2:	0	0	X	0	1	0
3:	1	1	1	0	X	0
4:	0	1	0	1	1	1
5:	0	0	1	1	0	1
6:	0	0	X	X	1	1
7:	1	X	X	1	0	0

■■

Test pattern reduction can be accomplished dynamically while patterns are being created.[21] In this approach the ATPG attempts to create tests for additional faults after a test has already been successfully created for a fault. In Figure 7.21 a test was created for the top input of gate *Q* SA1. This test was extended as far back as possible toward the inputs in an attempt to maximize fault comprehension. However, the PDCF for this fault immediately causes all paths from gates O and P to become "blocked"; that is, fault effects cannot propagate through those paths. However, in the circuit shown, gate *M* has fanout that leads to another primary output. It is possible that additional faults can be selected and sensitized to the other output. To do so would require selecting a fault and sensitizing a path to the other output, subject to the constraint that values must not be changed on gate inputs that have already been assigned. Values on those inputs are fixed and must not change.

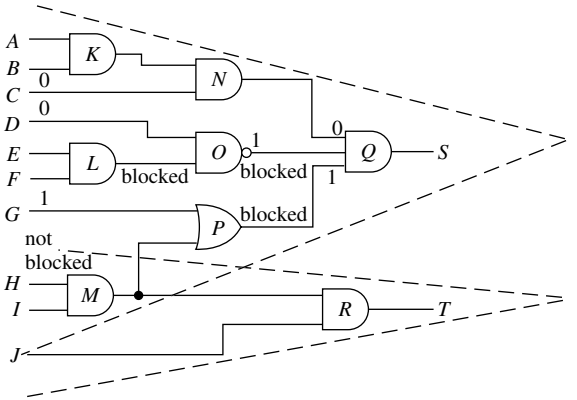


Figure 7.21 Dynamic compaction.

We attempt to propagate additional tests through gates that are not blocked. To increase the likelihood of selecting faults that can be successfully tested, cones are created from the outputs. Two cones are illustrated in Figure 7.21 by means of the dashed lines. Cones generally overlap since signals, especially control signals, affect many areas of logic. If a given fault is only contained in cones whose outputs already have assigned values, then it is pointless to select that fault during dynamic compaction.

In the circuit of Figure 7.21 a test on gate *K* could not be propagated to output *S* because it is blocked from the output. It cannot be propagated to output *T* because it is not contained in the cone of *T*. If an output has not yet had a value assigned, then a fault contained in the cone of that output is a candidate for test creation. If the test attempt fails because of excessive numbers of blocked gates, then continue until either a fault is found in that cone for which a test can be achieved or until no more untested faults exist for which a test has not been attempted. At some point in the creation of any one test pattern it becomes impractical to try to continue to create tests. Obviously, if all outputs in the circuit are assigned values, no additional faults can be propagated to these outputs. It also becomes difficult when nearly all of the inputs, ≥85%, have already been assigned values.

7.9.7 Test Counting

An interesting question, related to test compaction, is the following: "What is the smallest set of vectors needed to detect all of the stuck-at faults in a given circuit?" Consider the following expression, taken from the 74181 ALU in Section 7.8.2.

```
assign F3 = X3 ^ Y3 ^ (M | !(CN & X0 & X1 & X2 | Y0 & X1 & X2 | Y1 & X2 | Y2));
```

This expression is illustrated in Figure 7.22. A rather straightforward way to find a minimal test set would be to fault simulate all input combinations, then create a matrix of vector number versus faults detected. From that matrix it becomes a covering problem, much like the fault dictionaries discussed in Section 7.7.10; that is, find the smallest set of vectors that detects all faults. However, for large circuits with many inputs the matrix approach becomes impractical.

A small circuit such as the one in Figure 7.22 can be examined analytically without too much difficulty. First, note that some stuck-at faults can be tested in parallel. For example, if input *X*3 and the output of gate *F* are both 1s, then SA0 faults on either input to *G* will be detected at the same time. Therefore, tests for these faults can be readily merged with tests for other faults; thus for purposes of analysis, faults on these inputs can be ignored.

Tests for SA1 faults on AND gates *A*, *B*, and *C*, as well as on the inverter *D,* can exploit the fact that the tests do not block each other. However, each of the four inputs to gate *A* requires a separate test. Furthermore, a test for SA0 on each of the inputs to gate *E* requires a separate test. Hence, just from this brief, informal analysis it can be seen that there is a requirement for at least eight distinct test vectors for the circuit. In addition, a ninth vector is required to detect an SA0 on the input to *F* driven by input *M*, since *M* must be 0 in each of the preceding eight vectors to avoid blocking the propagation path from gate *E* to the output.

While the circuit in Figure 7.22 requires a minimum of nine vectors to detect all of its stuck-at faults, what is quite remarkable is the fact that the circuit in Figure 7.22 is but a very small piece of the circuit shown in Figure 7.23; it doesn't even include the selection logic used to generate *Xi* and *Yi*, and yet it has been shown that the circuit in Figure 7.23 can be fully tested with just 12 vectors.[22]

The method used to determine the number of vectors required to test the circuit of Figure 7.23 is called *test counting*. It does not compute the actual vectors needed to test the circuit, nor does it determine precisely how many vectors are needed. Rather, test counting derives a lower bound for the test counts. In order to determine the lower bound, some definitions are required. The *test values* 0^+, 1^+, 0^-, and 1^- are interpreted as follows: A 0^+ denotes a logic value 0 on a net that will detect an SA1

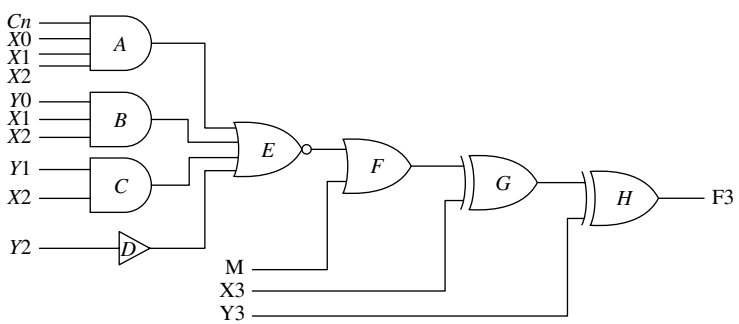


Figure 7.22 Circuit diagram for output F3.

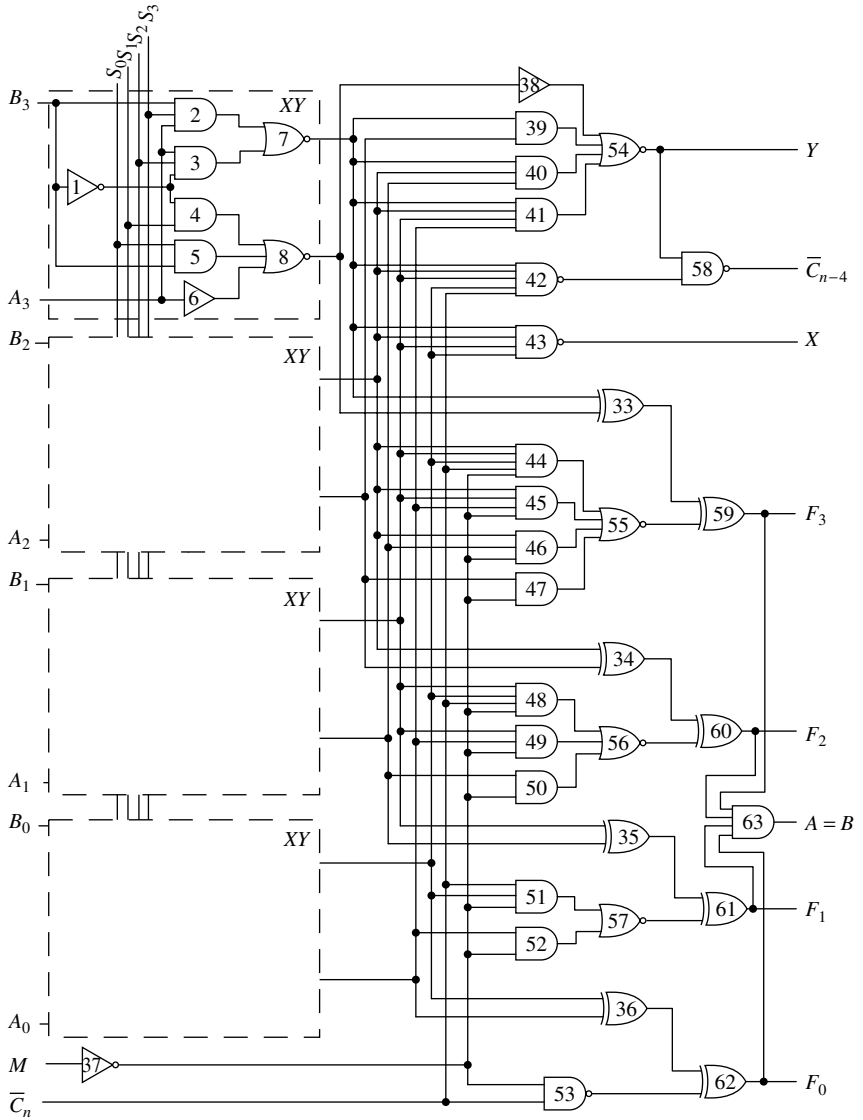


Figure 7.23 74181 ALU.

fault on that net. The net is sensitive to a SA1 fault. A 0^- denotes a logic 0 that will not detect an SA1 fault on the net. In this case the net is insensitive to the SA1 fault. The 1^+ and 1^- are interpreted analogously. The + and – are called *sensitivity values*. These values can be determined by simulating the circuit and identifying the sensitized paths reaching the output.

More interesting are the following values. They can be determined without simulating the circuit:

$$a_0^+, a_0^-, a_1^+, a_1^-, a_0, a_1, a^+, a^-$$

Given a circuit C, a set of test vectors T, and a net A, then

a_0^+ is the number of test vectors in T that produce a test value 0^+ on A

a_1 is the number of test vectors in T that produce a test value 1 on A

a^- is the number of test vectors in T that produce a sensitivity value – on A

The remaining five values are interpreted similarly. The symbol δ is used to represent the total number of test vectors in T.

In order to count the number of vectors required to test a combinational circuit, it is necessary to start with the basic building blocks, the logic elements. Consider a 4-input AND gate with inputs A, B, C, and D and output E. If (A, B, C, D, E) = (0^+, 1^-,1^-,1^-,0^+), this can be interpreted to mean that a 0 on input A will detect an SA1 on that input if the other inputs are nonblocking (i.e., logic 1), and an SA1 on the output of that gate is detectable at the output of the circuit. Put another way, the PDCF (0,1,1,1) on the inputs must propagate to the output. Since the test values on the inputs to the AND gate are mutually exclusive, the test count e_0^+ at the output E of the AND gate is 4.

Now, consider the circuit in Figure 7.23. We will informally analyze it to determine the number of test vectors needed to test for stuck-at faults on all the gate inputs. The interested reader can find a much more rigorous treatment of the subject in the original article.[22] The computations are performed by way of repeated passes through the circuit, until a complete pass through the circuit results in no more changes to any of the test values. However, in our simple circuit we will start at the inputs and, in one pass, compute all of the numbers. Note that at the output of gate A the value of a_0^+ is 4. The value of b_0^+ is 3, c_0^+ is 2 and d_0^+ is 1. These test values are not mutually exclusive; that is, the AND gates and the buffer can be tested in parallel. When these tests are propagated through the NOR labeled E, the test value e_1^+ becomes 4.

Testing the NOR is analogous to testing the AND. The test values (A, B, C, D, E) = (1^+, 0^-,0^-,0^-,0^+) are complementary, and the value of e_1^+ is 4. The values for the pair of test values (e_0^+, e_1^+) is (4, 4). The total number of tests required at this point in the circuit is the sum of the two numbers, or 8. One final calculation is required at the OR gate labeled F. This requires one additional vector, so the final result is $\delta = 9$. Recall that we determined that we could test the exclusive-OR gates in parallel with the tests coming from the preceding logic.

Note that this is a lower bound on the number of test vectors needed to test all of the modeled faults in the circuit. The test counts are computed without knowing what test vectors are applied to the circuit. In addition, the test count is affected by the choice of faults. Also note that when a circuit element fans out to two or more

elements, this must be taken into account. For example, if a stem *A* drives two checkpoint arcs *B* and *C* and the two arcs do not reconverge, then the possible values for (*A, B, C*) could be (0^+, 0^+, 0^+), (0^+, 0^+, 0^-), (0^+, 0^-, 0^+), (0^-, 0^-, 0^-), (1^+, 1^+, 1^+), (1^+, 1^+, 1^-), (1^+, 1^-, 1^+) or (1^-, 1^-, 1^-). If there is reconvergence, the number of possibilities increases and the computational complexity likewise increases.

Is there any value to test counting, or is it just an academic exercise? Given a scan-based circuit, it may be useful to know whether the number of vectors generated for a region of combinational logic is minimal or near minimal, since more vectors imply a longer test. That requires a greater amount of time on a tester, which adds to the cost of the die. In a large combinational array, test counting may prove useful in assessing the effectiveness of inserting test points at various places in the circuit to improve observability. Quantifying the improvement in vector count can help to make a more effective decision regarding cost of the test point versus cost of tester time for the die.

7.10 MISCELLANEOUS CONSIDERATIONS

A number of issues must be considered when developing a digital test plan. Some of these relate to design-for-testability (DFT) and will be postponed until the next chapter. However, other issues crop up during development of test programs or when evaluating different methodologies, and they must be resolved before test program development begins. We will examine some of those issues in this section. Before proceeding we note that, in the past, it was not uncommon for vendors to develop languages to control their fault simulators and/or ATPG programs. This is one of those areas that is giving way to standards: The Standard Test and Interface Language (STIL) discussed in Chapter 6 is not only suitable for the tester environment, it is sufficiently robust that it can also be used as an input medium for fault simulation and ATPG tools.

7.10.1 The ATPG/Fault Simulator Link

It was previously pointed out that an ATPG can be linked with a fault simulator under control of an executive routine. The ATPG creates sequences targeted at specific faults, and the fault simulator determines if the target fault was detected. In addition, the fault simulator identifies any other faults detected by the sequence passed on to it by the ATPG. If the sequence fails to detect the target fault and if no other faults are detected, the sequence is usually discarded. However, if any faults are detected, the sequence is retained and appended to the end of the test sequence.

Sequences can fail to accomplish their intended task for a number of reasons. The ATPG may simply have miscalculated. Often, when creating sequences targeted at a specific fault, the ATPG overlooks side effects that invalidate the sequence. One such problem is a failure to properly process bidirectional pins. The ATPG may attempt to drive a bidirectional pin with an external signal when the tri-state driver

for that I/O pad is active. A properly implemented fault simulator can recognize such conflicts and report the condition to the ATPG. This is especially important because the ATPG does not know how to deal with timing, and bidirectional pins frequently switch in the middle of a clock cycle.

Sometimes a sequence is invalidated by races or hazards. In the circuit depicted in Figure 7.24, two inputs switch at approximately the same time. As a result, there is the possibility of a negative-going pulse from the OR gate that may be of sufficient duration to cause the latch to make a permanent, but unintended, state transition. The transition, in turn, may block a fault effect from propagating forward to an output. A nominal delay fault simulator can identify race conditions that invalidate the work done by the ATPG. Not all states or input conditions cause problems. For example, the hazard depicted above will not cause an error if the output is already at logic 1.

If latches exist in a design, then vulnerable states must be identified. Requirements must be established for hazard-free signals on the inputs during the creation of a test pattern. This will reduce the freedom of choice on inputs to a logic gate. The hazard in Figure 7.24 may occur because of the manner in which justification is performed. If the ATPG simply requires that the output of the OR gate be at 1, then establishment of conditions for a 1 on the lower input to the OR gate would be deemed sufficient by the ATPG to satisfy the logic conditions imposed by the justification process. However, when an additional requirement is imposed that the net be hazard-free, the ATPG must consider previous assignments on the gate and determine if any hazard conditions are created as a result of the signal change. Furthermore, there may be a requirement that the circuit be free from exposure to dynamic as well as static hazards since a dynamic hazard on some circuits, such as a counter, can cause erratic counting operation.

A Delay flip-flop (DFF) must not be exposed to hazards on its Clock, Set, or Reset lines. A Data input may experience several changes during a clock period, but it is assumed that the data will stabilize before the clock is applied. For the cross-coupled NAND latch, the following requirements must hold:

$\overline{Set}$	$\overline{Reset}$	Q
1-1*	X-1	0-0
X-1	1-1*	1-1

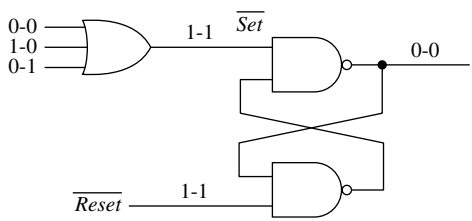


Figure 7.24 Occurrence of hazard.

In this table the first entry states that when the $\overline{Reset}$ goes from x to 1, the $\overline{Set}$ is at 1, and the latch is in state $Q = 0$, then the $\overline{Set}$ line must be free of hazards (an asterisk denotes the hazard-free requirement). In the second case, when in the state $Q = 1$, the $\overline{Reset}$ line must be hazard-free. Basically, any combination of internal state and input combination that could cause a state change in response to an unwanted pulse on an input line requires that the input line that is vulnerable to the pulse be hazard-free.

7.10.2 ATPG User Controls

Many design starts are so large that it is impractical to consider anything other than a full-scan test mode (cf. Chapter 8). However, there remain applications where the logic count is small enough that sequential ATPG can be considered. One such example might be battery operated human implants. Every effort is made to minimize gate count in such devices, so as to prolong battery life. The ability to control or influence operation of the ATPG can sometimes provide significant productivity enhancements in these situations. A *freeze pin* feature lets the user assign certain inputs to specific binary values, either for an entire run or for some specified number of vectors. A variation on that approach allows the user to specify certain combinations of input values that must be prohibited. Again, this could be for a fixed number of vectors or for an entire run. Input combinations can be prohibited if they cause transitions into illegal states or if they cause simultaneous toggling of either (a) clock and data inputs of a flip-flop or (b) load and clock inputs of a serial/parallel register. Other options may permit the user to include instructions on how to handle multiple clocks that require special sequencing.

When inputs are assigned a fixed value, these assignments are implicated by the ATPG and cause other logic gates to become blocked, just as during dynamic compaction. The same can be done for logic combinations on inputs. If two inputs are inhibited from being high simultaneously, then whenever one of them is set high by the ATPG, the other is immediately set low and all possible implications are performed.

The next logical extension of the concept of controlling or influencing the ATPG is the *guidance file*. This feature allows the user to provide a sequence of vectors that instruct the ATPG on how to drive the circuit into a particular state. By instructing the ATPG on how to navigate through complex control logic, such as state machines, which would otherwise be difficult to control, it may be possible for the ATPG to perform useful work in a circuit where it would otherwise simply thrash about unproductively. A particularly important area where the guidance file is useful is in those circuits that have convoluted initialization sequences. It is not unusual for an ATPG to fail completely on circuits where it could not compute the initialization circuit, but then produce useful results when the initialization sequence is provided by the user.

A potential pitfall in the use of guidance files is the fact that a bad, or incorrect, guidance file can be counterproductive. The ATPG may produce worse results than it

would without the guidance file. In addition, it could produce large numbers of vectors that do not increase fault coverage, but merely consume time on the tester. This is where the fault simulator can provide feedback, identifying sequences where the guidance file drove the circuit into an incorrect state.

7.10.3 Fault-List Management

The ability to manipulate fault lists is an important aspect of test program development. We saw earlier (Section 7.7.9) that a profiler tool can be very useful in identifying areas of a design where fault coverage is below acceptable levels. It may be desirable to target faults in those areas for special attention. When doing so, it may be more efficient if the fault list only contained faults from that part of the design being worked on. Otherwise, if an ATPG is being used, it may spend considerable CPU time pursuing faults from regions where fault coverage is already deemed acceptable. Other considerations must be taken into account; for example, there may be regions of the design that are to be tested using memory test or BIST. If a major function has dedicated BIST, then faults in that region of the design can be deleted from the fault list.

When several logic designers are working on a large circuit, they may be responsible for creating both the design verification vectors and the manufacturing test vectors for their part of the design. In such a case, they may prefer to run fault simulation strictly on those functions that are part of their responsibility. If the vectors they create have little effect on functions other than the one they are designing, then fault simulating other functions with their vectors will add little or nothing to overall fault coverage, but will slow down their fault simulation runs. In such cases a *merge fault* capability should be provided that can merge results from several designers into a master fault list.

The concept of granularity was discussed in Section 3.4. The general consensus in the test industry is that gate-level, stuck-at fault coverage gives acceptable results, consistent with the amount of CPU time required to fault simulate the test and the amount of tester time required to run the test. Occasionally, it may be desirable to fault simulate at the transistor level, but it will be costly in terms of CPU time if the circuit is very large, more than a few thousand gate equivalents.

Conversely, some users of fault simulation prefer to fault simulate at the macrocell level. They embed commands in library cells instructing the fault simulator to only fault the I/O ports. It is argued that stuck-at faults at a level of abstraction lower than cell I/O ports are speculative; that is, they cannot be shown to correspond to actual structural faults. However, test vectors can often provide 100% detection of port faults and still miss faults inside the cells. In fact, testing is an inexact science. Wadsack[8] describes an experiment where a device failed on a tester after the point in the vector file where the fault simulator reported 100% fault coverage. Yet, evidence suggests that, in general, fault coverage is better than the number predicted by the stuck-at model.

7.11 SUMMARY

In this chapter our purpose was to examine many different facets of test and tie them together into a comprehensive test strategy. Some methodologies have not yet been discussed, but at least with a clear picture of where we are, it is easier to go forward and determine how to fit other tools and strategies into the overall picture. It is also easier to make a judgment as to whether or not other tools are necessary and, if so, which tools can best help us reach our quality goals. Remember, in the final analysis the object is quality, not fault coverage. Fault coverage is a necessary, but not always sufficient, condition for quality. Fault coverage by itself may not guarantee protection against tester escapes, as was seen during discussion of the test triad at the beginning of this chapter.

Logic designers generate incredible amounts of intellectual property when creating test sequences to verify their designs. These vectors often accompany the design to the foundry, where they are used as the manufacturing test. Unfortunately, customers do not always fault simulate the vectors they send to the foundry. Several years ago, Texas Instruments was quoted as saying that 60% of their customers did not perform fault grading.[23] That is risky because the vectors serve as an acceptance criteria. If the fault coverage provided by the vectors is low, the customer receives chips from the foundry whose quality is suspect.

Fault modeling is an important aspect of test program development. It is important to model at a level of detail that gives meaningful results while ensuring that fault simulation runs complete in a reasonable amount of time. Good fault management tools are critical to this effort. They should allow a test development team to focus their efforts in a way that maximizes productivity. The tools should also facilitate maximum leverage of test programs generated for design verification. Even in those cases where an ATPG is used, design verification vectors can be useful if first silicon does not function as intended on the tester. A logic designer may be completely befuddled by test vectors that were created by an ATPG. That same designer is often able to quickly diagnose and debug failures that occur while the tester is running vectors that he created.

Test vectors are often created in similar ways, whether intended for design verification or for manufacturing test. A major difference is that the designer, when checking out a design, often uses functions that have already been thoroughly debugged and checked out, so he or she only wants to make sure that the function has been properly connected into a larger design. However, when creating a test whose purpose is to detect physical defects, it is necessary to exercise all functionality in the design.

Despite significant amounts of research into behavioral fault simulation, it is still performed primarily at the gate level using the stuck-at fault model. This is so because the approach works; that is, fault coverage provided by the stuck-at fault model is reasonably accurate, based on three decades of experience, and because no other approach offers a compelling reason to replace the existing system. One area where it would seem that the industry could benefit from the behavioral or functional approach is in the development of algorithmic test programs for standard functions. Some

functions lend themselves nicely to algorithmic test program development, in a sense analogous to the test programs that have evolved over the years for memory tests.

PROBLEMS

7.1 Derive PDCFs and propagation cubes from the truth table (a) in the example in Section 7.5.1.

7.2 For the stem driven by gate Q in Figure 7.4, find vectors that detect the checkpoint faults emanating from that stem but do not detect the stem fault.

7.3 List all of the checkpoint and stem faults for the circuit in Figure 4.1. Collapse this list to get a minimal list of faults for the circuit. Starting at the Inputs, I_1, ..., I_5, how many unique signal paths from inputs to outputs can you identify?

7.4 Using the circuit in Figure 4.1, verify that the pattern (I_1, I_2, I_3, I_4, I_5) = (0, 0, 1, 0, 0) detects SA0 faults on inputs to gates I and L, but not on the output of gate D. Identify all faults detected by that pattern.

7.5 The circuit in Figure 4.1 has a redundant input on gate G. Which input is it? Explain your answer.

7.6 Identify all faults in the NOR circuit that are detected by the six test vectors developed for the NAND circuit of Figure 7.9. Create a pass–fail vector for each fault and use that to create a fault dictionary. Which two NOR gates could be completely missing from the circuit and fail to be detected by the test sequence given in Section 7.6.4?

7.7 Given the expression $Y = A \cdot B + C \cdot D \cdot E + F$; if the vectors A,B,C,D,E = {000000, 010000, 001100} were applied to the circuit, what is the total expression coverage for the circuit?

7.8 The critical path was described in Section 4.6.3. Explain how you would apply it to the circuit in Figure 7.22 in order to get a minimum set of test vectors.

7.9 For the circuit of Figure 7.23, generate a minimum set of vectors that will detect all faults in the cone of output F_3.

7.10 For the circuit in Figure 7.22:

(i) Change the function of gate A to a NAND. Then compute the number of vectors required to test all of the stuck-at faults.

(ii) Assume that gate A is an eight-input AND gate. Then what is the minimum number of vectors required to test all of the stuck-at faults?

7.11 Using the circuit in Figure 7.18, create the smallest possible complete test set for

(a) Maximum resolution

(b) Maximum comprehension

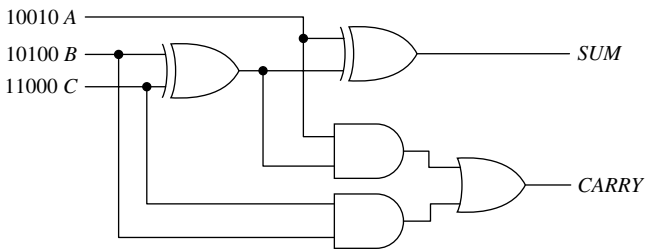


Figure 7.25 Full-adder circuit.

7.12 In the full-adder circuit of Figure 7.25, the five vectors (one per column) will detect all port faults. Find a stuck-at fault inside the macrocell that is not detected by the five vectors.

7.13 Given the following two-input logic gates, with values on the inputs and output as indicated, which of the assignments imply additional values?

	In1	IN2	OUT
OR	x	x	0
OR	1	x	x
NOR	x	x	1
NAND	1	x	1
AND	0	x	0

7.14 Given the following matrix of test patterns versus faults detected:

		Fault Number							
		1	2	3	4	5	6	7	8
	1	1		1			1		
Pattern	2		1	1	1				
Number	3			1		1		1	
	4			1				1	1

(a) If pattern 4 is the only failure, which fault is most likely to have occurred?

(b) If all four tests fail, which fault is most likely to have occurred?

7.15 Use static compaction to minimize the following set of vectors: {01X0X0, 10X0X0, X001X0, X010X0, X0X001, X0X010, 11X0X0, X0X0X0, 11X0X0, X011X0, X0X011}.

7.16 Merge the following three sequences of patterns:

1 1 1 X X 0	X 1 X X 1 0	1 1 0 0 1 1
X 1 X 1 1 X	X 0 X 1 X 1	1 X 1 1 1 0
0 0 X X 0 1	X 0 0 0 0 1	0 0 0 X X 0
1 X X 0 X 1	1 1 1 X 1 0	1 X X X 1 0
		1 1 1 0 1 0

7.17 Find a sequence of four tests that will detect all seven CMOS NOR gate faults.

7.18 Explain how you would create a four-vector set that provides 100% fault coverage for a parity checker of arbitrary size $n > 0$.

7.19 Using the Apply and Reduce algorithms (cf. Section 2.11), create BDDs for the two circuits in Figure 7.9 and show that they are equivalent.

7.20 Use test counting to find a minimum set of vectors that detect all the stuck-at faults in the circuit of Figure 4.1.

7.21 If you have access to a commercial logic synthesis program and fault simulator, synthesize the 16-bit counter b16ctr given in Section 7.8.2 and fault simulate it using the test bench given in that same section.

7.22 Repeat the previous problem, using the algorithmic test for an ALU and n copies of the 74181 (or other ALU).

REFERENCES

1. Maxwell, P. C., R. C. Aitken, V. Johansen, and I. Chiang, The Effectiveness of I_{DDQ}, Functional and Scan Tests: How Many Fault Coverages Do We Need?, *Proc. Int. Test Conf.*, October 1992, pp. 168–177.
2. Goel, P., J. Grason, and D. Siewiorek, Structural Factors in Fault Dominance for Combinational Logic Circuits, *Proc. Fault Tolerant Comput. Symp.*, 1971.
3. Armstrong, D. B., On Finding a Nearly Minimal Set of Fault Detection Tests for Combinational Logic Nets, *IEEE Trans. Electron. Comput.*, Vol. EC-15, No. 1, February 1966, pp. 66–73.
4. Mei, K. C. Y, Bridging and Stuck-at Faults, *IEEE Trans. Comput.*, Vol. C-23, No. 7, July 1974, pp. 720–727.
5. Son, K., and D. K Pradhan, Design of Programmable Logic Arrays for Testability, *Proc. IEEE Test Conf.*, 1980, pp. 163–166.
6. Szygenda, S. A., and A. A. Lekkos, Integrated Techniques for Functional and Gate-Level Digital Logic Simulation, *Proc. 10th Des. Autom. Conf.*, pp. 159–172.
7. Thomas, J. J., Common Misconceptions in Digital Test Generation, *Comput. Des.*, January 1977, pp. 89–94.
8. Wadsack, R. L., Fault Modelling and Logic Simulation of CMOS and MOS Integrated Circuits, *Bell Syst. Tech. J.*, Vol. 57, No. 5, May–June 1978, pp. 1449–1474.

9. El Ziq, Y. M., Automatic Test Generation for Stuck-Open Faults in CMOS VLSI, *Proc. 18th D.A. Conf.*, 1981, pp. 347–354.
10. Beh, C. C. et al., Do Stuck Fault Models Reflect Manufacturing Defects?, *Proc. IEEE Test Conf.,* 1982, pp. 35–42.
11. Wadsack, R. L., Fault Coverage in Digital Integrated Circuits, *Bell Syst. Tech. J.*, May–June 1978, pp. 1475–1488.
12. Miczo, A., Fault Modelling for Functional Primitives, *Proc. IEEE Test Conf.*, 1982, pp. 43–49.
13. Wadsack, R. L., Design Verification and Testing of the WE 32,100 CPUs, *IEEE Des. Test*, August 1984, pp. 66–75.
14. Maxwell, Peter C., Reductions in Quality Caused by Uneven Fault Coverage of Different Areas of an Integrated Circuit, *IEEE Trans. Comput.-Aided Des. Int. Circuits Syst.*, Vol. 14, No. 5, May 1995, pp. 603–607.
15. Chang, H. Y., E. Manning, and G. Metze, Fault Dictionaries, *Fault Diagnosis of Digital Systems*, Chapter 5, John Wiley & Sons, New York, 1970.
16. Megill, N., Techniques for Reducing Pattern Count for Functional Testing, *Proc. 1979 Int. Test Conf.*, pp. 90–94.
17. Miczo, A., Enhanced Test Through Improved RTL Code Coverage, *Proc. High Level Des. Validation & Test Workshop*, November 1997, pp. 99–104.
18. Snethen, T. J., Simulator-Oriented Fault Test Generator, *Proc. 14th D.A. Conf.*, 1977, pp. 88–93.
19. Roth, J. P., Diagnosis of Automata Failures: A Calculus and a Method, *IBM J. Res. Dev.*, Vol. 10, No. 4, July 1966, pp. 278–291.
20. Roth, J. P. et al., Programmed Algorithms to Compute Tests to Detect and Distinguish Between Failures in Logic Circuits, *IEEE Trans. Electron. Comput.*, Vol. EC-16, No. 5, October 1967, pp. 567–580.
21. Goel, P., and Barry C. Rosales, Test Generation & Dynamic Compaction of Tests, *Proc. 1979 Int. Test Conf.*, pp. 189–192.
22. Akers, S. B., and B. Krishnamurthy, Test Counting: A Tool for VLSI Testing, *IEEE Des. & Test of Computers*, Vol. 6, No. 5, October 1989, pp. 58–77.
23. Anonymous, *Comput. Des.*, April 1, 1991, p. 64.

CHAPTER 8

Design-For-Testability

8.1 INTRODUCTION

Chapter 7 focused on methods for integrating design and test activities by capturing verification suites written by logic designers and converting them to test programs. For some ICs, especially those with reasonably high yield, test programs derived from a thorough design verification suite, combined with an I_{DDQ} test (cf. Chapter 11), may produce quality levels that meet or exceed corporate requirements.

When it is not possible, or practical, to achieve fault coverage that satisfies acceptable quality levels (AQL) through the use of design verification suites, an alternative is to use an automatic test pattern generator (ATPG). Ideally, one would like to reach fault coverage goals merely by pushing a button. That, however, is not consistent with existing state of the art. It was pointed out in Chapter 4 that several ATPG algorithms can, in theory at least, create a test for any fault in combinational logic for which a test exists. In practice, even when a test exists for a large block of combinational logic, such as an array multiplier, the ATPG may fail to generate a test because of the sheer volume of data that must be manipulated.

However, the real stumbling block for ATPG has been sequential logic. Because of the inability of ATPGs to successfully deal with sequential logic, a growing number of digital designs are being designed in compliance with formal design-for-testability (DFT) rules. The purpose of the rules is to reduce the complexity of the test problem. DFT guidelines prohibit design practices that impede testability, and they usually call for the insertion of special constructs into designs solely to facilitate improved testability. The focus over the past two decades has shifted from testing function to testing structure. As an additional benefit, testable designs are frequently easier to design and debug. The design restrictions that make it easier to generate test programs also tend to prohibit design practices that introduce difficult to diagnose design errors. The payback is not only higher quality, but also faster time-to-volume; in addition, fault coverage requirements are achieved much sooner, and products reach the marketplace sooner.

Digital Logic Testing and Simulation, *Second Edition*, by Alexander Miczo
ISBN 0-471-43995-9 Copyright © 2003 John Wiley & Sons, Inc.

8.2 AD HOC DESIGN-FOR-TESTABILITY RULES

When small-scale integration (SSI), medium-scale integration (MSI), and large-scale integration (LSI) were the dominant levels of component integration, large systems were often partitioned so that data flow paths and control circuits were placed on separate printed circuit boards (PCBs). Most PCBs in a given design contained data flow circuits that were not difficult to test using an ATPG. A lesser number contained the more complex control logic and handshaking protocols. Test programs for control logic would be created by requiring a logic designer or test engineer to write vectors that were then fault simulated to determine their effectiveness. Since the complex PCBs made up a smaller percentage of the total, test creation was not excessively labor-intensive. The task of writing tests for these boards was further simplified by the fact that sequential transitions in control logic could often be observed directly at I/O pins rather than indirectly through observation of their effects on data flow logic.

The evolution of technology has brought about an era where individual ICs now possess hundreds of thousands to millions of gates. RAM and ROM often reside on the same IC with complex logic. Individual I/O pins serve multiple purposes, acting both as inputs and as outputs. The increasing gate to pin ratio results in fewer I/O pins with which to gain access to the logic to be tested. Architecturally, many chips have complex arbitration sequences that require several exchanges of signals before anything meaningful happens inside the chip. All of these factors contribute to potentially long test programs that strain the resources of available test equipment and point to the conclusion that test issues must be considered early in the design cycle.

It was pointed out in Section 1.2 that acceptable quality level (AQL) is a function of both the process yield and the thoroughness of the test program. If the process yield is high enough for a given product, it may not need a test, only an occasional sampling to ensure that processing steps remain within tolerances. Consider an IC for a digital wristwatch. It could be very expensive to test every chip for all stuck-at faults. But the yield on such chips is high enough that an occasional sampling of ICs is adequate to ensure that they will function correctly; and if an occasional defective IC slips through the screening process unnoticed, it is not likely to have severe economic consequences.

Ad hoc DFT addresses circuit configurations that make it difficult or impossible to create effective test programs, or cause excessively long test sequences. The adverse effects of these circuit configurations may be local, affecting only a few logic elements, or they may be global, wherein a single circuit construct causes an IC or PCB to become completely untestable. Some problems may manifest themselves only under adverse environmental conditions—for example, temperature extremes, humidity, physical vibrations, and so on. A solution to a particular problem is sometimes quite simple and straightforward, the most difficult part of the problem being the recognition that there is a problem.

Testability problems for digital circuits can be classified as controllability or observability problems (or both). *Controllability* is a measure of the ease or difficulty with which a net can be driven to a known logic state. *Observability* is a measure of

the ease or difficulty with which a logic value on a net can be driven to an output where it can be measured. Note that observability is often a function of controllability, meaning that it may be impossible to observe a given internal node if the circuit cannot be driven to (i.e., controlled to) a given state. Expressed in terms of controllability and observability, the goal of DFT is to make the behavior of a circuit easier to control and observe.

We begin by looking at some circuit configurations that cause problems in digital circuits. That will be followed by an examination of techniques used to improve controllability and observability. The solutions are often rather straightforward, and frequently there is more than one solution, in which case the solution chosen will depend on the resources available, such as the amount of board or die space and/or number of edge pins. Ad hoc solutions target specific test problems uncovered during the design and test process, and in fact similar test problems may be solved quite differently on different projects. In later sections we will look at formal methods for DFT. A *formal* DFT methodology, as used in this text, refers to a methodology that is well-defined, rigorous, and thorough. It is usually adopted at the very beginning of a project.

8.2.1 Some Testability Problems

Design practices that adversely affect controllability and observability are best understood in terms of the difficulties they create for simulation and ATPG software. It is not possible to list all of the design practices that cause testing difficulties, since some practices may be harmless in one application, yet detrimental in another. The emphasis will be on understanding why certain practices create untestable designs so the designer can exercise some judgment when uncertain about whether a particular design practice causes problems.

In the past, when many PCBs were designed using SSI, MSI, and LSI, in-circuit testers were commonly used as the first testing station, because they could quickly find many obvious errors such as ICs mounted incorrectly on the PCB, the wrong IC in a particular slot, IC pins failing to make contact with metal runs, or solder shorts between pins (cf. Section 6.6). However, in those applications where the in-circuit tester is used, design practices can reduce its effectiveness. In-circuit testers access tests from a standard library of tests and apply those tests to components on a PCB. These tests make assumptions about controllability and observability of I/O pins on the devices. If a device cannot be controlled and if the test cannot be modified or a new test obtained, then the device cannot be tested.

Unused IC signals such as chip-select and output-enable are usually tied to an enabling state. For example, a common practice in PCB design is to tie unused inputs of Delay and J-K flip-flops directly to ground or power. This is especially true for Set and Clear lines on discrete flip-flops in those applications where they are not required to be initialized at system start-up time. This practice impedes the ability of the in-circuit tester to control the device. If an in-circuit tester is used as part of the test strategy for a PCB, unused pins that must be controlled during test should be tied to power or ground through a resistor.

Disabled Set and Clear lines cause further problems when a flip-flop is used as a frequency divider. In Figure 8.1 an oscillator driving toggle flip-flops presents a problem for test because its operating frequency may be known but not its phase. At a given point in time, is it rising or falling? For test purposes, the oscillator must be controlled. However, even when it is controlled, the circuit presents problems. Two clock pulses at a toggle input generate one pulse at its output, producing a frequency divider. Two or more toggle flip-flops can be tied in series to further reduce the main clock frequency. The value at the output of the divider circuit is not known at any given time, nor does it need to be known for correct operation of the circuit, since other handshaking signals are used to synchronize the exchange of data between devices clocked at different frequencies. What is known is that the output will switch at a fraction of the main clock frequency, and therefore some device(s) will be clocked at the lower rate.

A frequency divider can produce the usual problems associated with indeterminate states for simulation and test. However, even when the correct state can be determined, if several frequency divider stages are connected in series, then a large number of input patterns must be applied to cause a single change at the output of the frequency divider. These patterns can require exorbitant amounts of CPU time to simulate and, worse still, exorbitant amounts of time on a tester.

Several methods exist for creating pulse generators in sequential circuits and virtually all of them cause problems for ATPG programs. The methods include use of single shots, also known as self-resetting flip-flops, as well as circuits that gate a signal with a delayed version of that same signal. The single-shot is shown in Figure 8.2(a), and the gated signal is shown in Figure 8.2(b). A correct and complete description of the behavior of either of these circuits requires the use of the time domain. A logic event occurs but persists only for some brief elapsed time, after which the circuit reverts to its previous state. However, ATPGs generally see only the logic domain, they do not recognize the time domain. When the ATPG clocks the single-shot, the 0 at $\overline{Q}$ will eventually reset the flip-flop. But, since the ATPG does not recognize the passage of time, it will conclude that the flip-flop immediately returns to 0. Similar considerations hold for the circuit of Figure 8.2(b).

Another problem is presented by the circuit in 8.2(a). Generally, an ATPG considers storage elements to be in the indeterminate state when power is first applied. As a result, the Q and $\overline{Q}$ outputs are initially set to x, and that causes an x to appear at the $\overline{\text{Reset}}$ input. If the ATPG attempts to clock a logic 1 through the flip-flop and

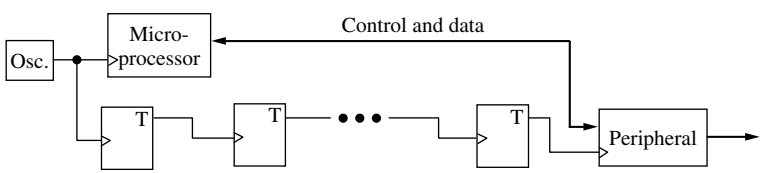


Figure 8.1 Peripheral clocked by frequency divider.

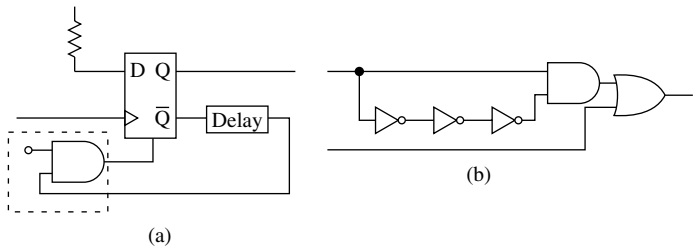


Figure 8.2 Pulse generators.

sees the x on the $\overline{\text{Reset}}$ input, it will leave the flip-flop in the x state. Note that since the circuit will settle in a known state, a dummy AND gate can be added to the circuit to force the circuit model to assume that known state.

An important distinction between this circuit and the frequency divider is the fact that it is known how the self-resetting flip-flop behaves when power is applied. If it comes up with $Q = 0$, then it is in a stable state. If Q is initially a 1 following application of power, then the 0 on $\overline{Q}$ causes it to reset. Therefore, regardless of the initial state, it is predictably in a 0 state within a few nanoseconds after power is applied.

When the state of a device can be determined, the ATPG or simulator can be given an assist. In this case, any of the following three methods can be used:

1. Model the circuit as a primitive (a monostable).
2. Specify an initial state for the circuit.
3. Use a dummy reset.

If the circuit is modeled as a primitive, then a pulse on the clock input to this primitive causes an output pulse of some duration determined by the delay. Allowing the user to specify an initial state, or using a special ATPG cell in a library, can solve the problem, since either value causes it to achieve a stable state. However, if an indeterminate logic value should reach the clock line at a later point in time, it could cause the circuit to revert to the indeterminate state.

In combinational logic, when many signals converge at a single node, such as when an AND gate has many inputs, then observability of fault symptoms along any individual path converging on that gate requires setting all other inputs to 1 (the nonblocking value). If this node in turn fans out to several other gates, then controllability of those gates is diminished in proportion to the difficulty in setting the convergent node to a 0 or 1. An AND gate with n inputs recognizes 2^n input combinations. All but 1 of those combinations produces a 0 at the output. If even a single input is difficult to set to 1, that input can block a test path for all other inputs. If the output of the AND gate fans out to other logic, that one gate affects observability of logic up to that point and it affects controllability of logic following that node.

An 8-bit bus may carry a 7-bit ASCII code together with a parity bit intended to produce even parity. The parity checker may be designed so that its output is normally low unless some fault causes odd parity to occur on the bus. But some faults in the parity checker may inhibit it from going high. To detect these faults, it must be possible to get odd parity on the 8-bit bus, but the bus is designed to generate even parity. Hence a test input to the parity checker is required or the parity generator that creates the bus parity bit must be controllable independent of its parity-generating logic.

Counters, like frequency dividers, can cause serious test problems because a counter with n stages may require up to 2^n clocks to drive it into a particular state if it does not have a parallel load capability. If the counter has a serial load capability, then any value can be loaded into it in n clock steps. Some other design practices that cause test problems include the following:

- Connecting drivers in parallel to get more drive capability
- Randomly assigning unused states in state machines
- Gating clock lines with data signals

Parallel drivers are a problem because if one of the drivers should fail, the result may be an intermittent error whose occurrence depends on unpredictable environmental factors and internal operating conditions. Repeating the problem for the purposes of diagnosis and repair becomes almost impossible under such conditions.

Unused states in a state machine are often assigned so as to minimize logic. As a result, an erroneous transition into an unassigned state, followed by a transition to a valid state, may go undetected but cause data corruption. The severity of the problem depends on the application. To err on the side of safety, a transition into an illegal state should normally cause some noticeable symptom such as an error signal or, at the very least, continued transitions into the same illegal state, that is, a "hangup," so an operator can detect the presence of the malfunction before serious damage is done by the device. Transitions into incorrect states can occur when hazards cause unintended pulses on clock lines of flip-flops. One way to avoid this is to avoid gating clock signals with data signals. This can be done by using the data signal that would be used to gate the clock to control a multiplexer instead, as shown in Figure 8.3. The Load signal that the designer might have used to gate the clock is used instead to either select new data for input to the flip-flop or to hold the present state of the flip-flop.

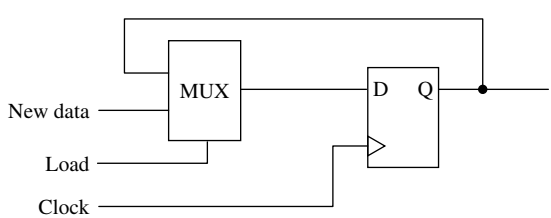


Figure 8.3 Load enable for flip-flop.

8.2.2 Some Ad Hoc Solutions

The most obvious approach to solving observability problems is to connect a tester directly to the output of a gate that has poor observability. Since that is quite impractical in dense ICs, methods have been devised over the years to employ functional I/O pins during test. Troublesome internal circuits can be routed to these pins in order to improve testability. A major problem with this approach is the cost of I/O pins. Design teams are reluctant to cede these pins to the solution of test problems. However, as feature sizes continue to shrink, more real estate becomes available on the die, and logic becomes available to permit the sharing of I/O pins (cf. Section 8.4).

If a particular region of an IC has low observability, it is possible to route several internal nodes to an output through an observability tree, depicted in the dashed lines in Figure 8.4. Several signals can be directly observed, and symptoms do not become blocked or transformed by other logic.

Note that the observability tree connects four internal signals to a parity tree whose output drives an I/O pin. If an error signal appears at any one (or an odd number) of parity tree inputs, the parity tree output will have the wrong value and the fault will be detected. Many faults can simultaneously produce error signals at the inputs to the parity tree and become detected, just as they would at any other I/O pin. If a fault causes error signals to appear at two, or an even multiple, of parity tree inputs, the signals will cancel out and the fault will escape detection. That, however, is highly improbable, and even more unlikely to occur on many vectors. The parity tree shown here has four inputs, but, in practice, the number of inputs is limited only by practical concerns. For each multiple of two, the depth of the parity tree increases one level. So, a 32-input parity tree will be five levels deep. The depth must be taken into consideration since it might exceed the clock period of the circuit.

Internal nodes that should be connected to the parity tree inputs shown in Figure 8.4 can be selected by means of fault simulation. The fault simulator is run with a fault list consisting only of undetected faults. If the fault simulator is instrumented to observe the nodes at which error signals appear, it can maintain a count at each of these nodes. Since all of the error signals emanate from undetected faults, the count of unique fault effects passing through a given node is a measure of the number of undetected faults that could be detected if that node were made to be observable.

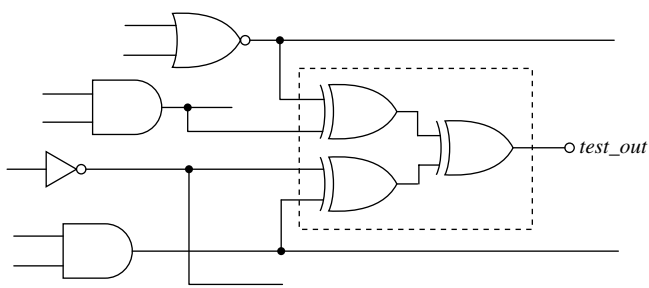


Figure 8.4 Observability enhancement.

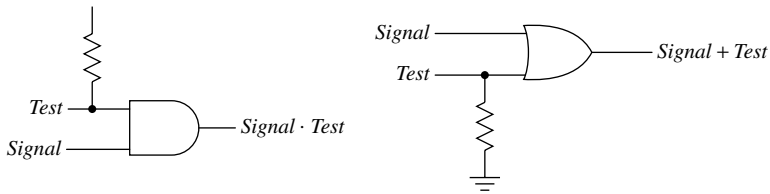


Figure 8.5 Controllability for 1 or 0 state.

At the conclusion of fault simulation, the nodes can be ranked based on the number of undetected faults observed at each node. Note, however, that if n_1 faults are observed at node N_1, and n_2 faults are observed at node N_2, the total T_d of faults that become detectable by making both nodes observable is $T_d \leq n_1 + n_2$ because some of the undetected faults may be included in the count for each of the two nodes. Because observability tends to be rather uneven across an IC, many undetected faults often are clustered together in a local area. Hence, this observability enhancement can be quite effective when targeted at regions of the circuit that have low observability.

Controllability can be improved by adding an OR gate or an AND gate to a circuit, together with additional I/O pins. The choice depends on whether the difficulty lies in obtaining a logic 0 or logic 1 state. The logic designer may be aware, either from a testability analysis tool or from a basic understanding of the circuit, that the 0 state is easily obtained but that setting up the 1 state requires an elaborate sequence of state transitions occurring in strict chronological order. In that case a two-input OR gate is used. One input comes from the net that is difficult to control, and the other input is tied to an edge pin. In normal use the input is grounded through a pull down resistor; during testing the input is pulled up to the logic 1 state when that value is needed. Where the logic 0 is difficult to obtain, an AND gate is used.

If the test environment, including the technology and packaging, permit direct access to the IC pins, then the edge pin connection can be eliminated. The IC pin is tied only to pull-up or pull-down resistors, as in Figure 8.5, and the tester is placed directly in contact with the IC pin by some means.

If both logic values must be controlled, then two gates are used, as illustrated in Figure 8.6(a). The first gate inhibits the normal signal when its test input is brought low, and the second gate is used to insert the desired test signal. This configuration gives complete control of the signal appearing on the net for both the 0 and 1 states

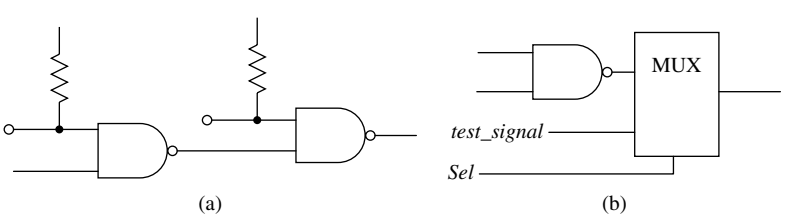


Figure 8.6 Total controllability.

at the cost of two I/O pins and two gates. The inhibit signal for several such circuits can be connected to a single I/O pin, to reduce the number of edge pins required. This configuration can be implemented without I/O pins if the tester can be connected directly to the IC pins; otherwise a multiplexer can be used, with the *Sel* signal used to choose the source. If switches are allowed on the PCB, then controllability of the net can be achieved by replacing the multiplexer with a switch.

Total controllability and observability at a troublesome net can be achieved by bringing the net to a pair of edge pins, as shown in Figure 8.7(a). These pins are reconnected at the card slot. This solution may, of course, create its own problems if the extra wire length picks up noise or adds excessive delay to the signal path. An alternate circuit, shown in Figure 8.7(b), uses a tri-state gate. In normal operation the tri-state control is held at its active state and the bidirectional I/O pin is unused. During test, the bidirectional pin is used to observe logic values when the tri-state control is active or to inject signals when the tri-state disables the output of the preceding gate. A single tri-state control can disable several gates to minimize the number of I/O pins required.

Some additional solutions, where possible, to testability problems include the following:[1]

- Use sockets for complex devices such as microprocessors and peripherals.
- Make memory read/write lines accessible at a board edge pin.
- Buffer the primary inputs to a circuit.
- Put analog devices on separate boards.
- Use removable jumper wires.
- Employ standard packaging.
- Provide good documentation.

As explained in Chapter 6, automatic test equipment (ATE) usually has different drive characteristics from the devices that will drive primary input pins during normal operation. If devices are connected directly to primary input pins without buffering, critical timing relationships between the signals may not be maintained by the ATE.

Analog devices, such as analog-to-digital and digital-to-analog converters, usually must be tested functionally over their entire range. This becomes exceedingly difficult when they are on the same board with digital logic. Voltage regulators placed on a board with digital logic can, if performing marginally, produce many seemingly different and unrelated symptoms within the digital logic, thus making diagnosis more difficult.

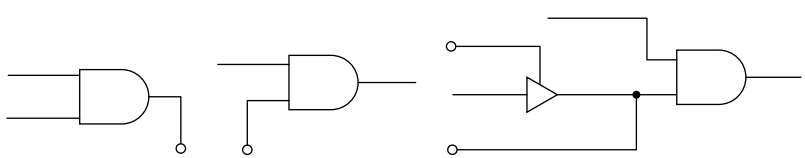


Figure 8.7 Total controllability and observability.

Finally, some practical considerations to aid in diagnosis of faults can provide a substantial return on investment. Removable jumper wires may significantly reduce the amount of time required to diagnose failures. Standard packaging, common orientation, spacing and numbering can reduce error and confusion during troubleshooting. Good documentation can be invaluable when trying to diagnose the cause of a failure.

8.3 CONTROLLABILITY/OBSERVABILITY ANALYSIS

In the previous section we described some techniques for solving particular testability problems. Some of the configurations virtually always create test problems. Other circuit configurations are not problems in and of themselves but can become problems when they appear in excessive numbers. A small number of flip-flops, connected in a straightforward manner without feedback, apart from that which exists inside the flip-flops, and without critical timing dependencies, can be relatively easy to test. Testability problems occur when large numbers of flip-flops are connected in serial strings such that control of each flip-flop depends on first controlling its predecessors in the chain. Examples that we have seen include the counter and the frequency divider.

Fortunately, the counter and frequency divider are reasonably easy to recognize. In many circuits the nodes that are difficult to test are not so easy to identify. For example, an AND gate may be controlled by several signals and it, in turn, may control several other logic gates. The node may be a problem or it may, in fact, be rather easy to test. Programs for measuring testability have been developed that help to determine which nodes are most likely to be problems.

8.3.1 SCOAP

SCOAP (Sandia Controllability Observability Analysis Program) is a testability analysis program that assigns numbers to nodes in a circuit.[2] The numbers reflect the relative ease or difficulty with which internal nodes can be controlled or observed, with higher numbers being assigned to nodes that are more difficult to control or observe. The program computes both combinational and sequential controllability and observability numbers for each node; furthermore, controllability is broken down into 0-controllability and 1-controllability, recognizing the fact that it may be relatively easy to generate one of the states at the output of a logic gate while the other state may be difficult to produce. For example, to get a 0 on the output of an AND gate requires a 0 on any single input. However, to get a 1 on the output requires that 1s be applied to all inputs. That, in general, will be more difficult for gates with larger numbers of inputs. Because observability depends on controllability, the controllability equations will be discussed first.

The Controllability Equations The e-controllability, $e \in \{0,1\}$, of a node depends on the function of the logic element driving the node and the controllability of the inputs to that element. If the inputs are difficult to control, the output of that

function will be difficult to control. In a similar vein, the observability of a node depends on the elements through which its signals must propagate to reach an output. Its observability can be no better than the observability of the elements through which it must be driven. Therefore, before applying the SCOAP algorithm to a circuit, it is necessary to have, for each primitive that appears in a circuit, equations expressing the 0- and 1-controllability of its output in terms of the controllability of its inputs, and it is necessary to have equations that express the observability of each input in terms of both the observability of that element and the controllability of some or all of its other inputs.

Consider the three-input AND gate. To get a 1 on the output, all three inputs must be set to 1. Hence, controllability of the output to a 1 state is a function of the controllability of all three inputs. To produce a 0 on the output requires only that a single input be at 0; thus there are three choices and, if there exists some quantitative measure indicating the relative ease or difficulty of controlling each of these three inputs, then it is reasonable to select the input that is easiest to control in order to establish a 0 on the output. Therefore, the combinational 1- and 0-controllabilities, $CC^1(Y)$ and $CC^0(Y)$, of a three-input AND gate with inputs X_1, X_2 and X_3 and output Y can be defined as

$$CC^1(Y) = CC^1(X_1) + CC^1(X_2) + CC^1(X_3) + 1$$
$$CC^0(\mathrm{Y}) = \mathrm{Min}\{CC^0(X_1), CC^0(X_2), CC^0(X_3)\} + 1$$

Controllability to 1 is additive over all inputs and to 0 it is the minimum over all inputs. In either case the result is incremented by 1 so that, for intermediate nodes, the number reflects, at least in part, distance (measured in numbers of gates) to primary inputs and outputs. The controllability equations for any combinational function can be determined from either its truth table or its cover. If two or more inputs must be controlled to 0 or 1 values in order to produce the value e, $e \in \{0,1\}$, then the controllabilities of these inputs are summed and the result is incremented by 1. If more than one input combination produces the value e, then the controllability number is the minimum over all such combinations.

Example For the two-input exclusive-OR the truth table is

X_1	X_2	Y
0	0	0
0	1	1
1	0	1
1	1	0

The combinational controllability equations are

$$CC^0(Y) = \mathrm{Min}\{CC^0(X_1) + CC^0(X_2), CC^1(X_1) + CC^1(X_2)\} + 1$$
$$CC^1(\mathrm{Y}) = \mathrm{Min}\{CC^0(X_1) + CC^1(X_2), CC^1(X_1) + CC^0(X_2)\} + 1$$ ■■

The sequential 0- and 1-controllabilities for combinational circuits, denoted SC^0 and SC^1, are computed using similar equations.

Example For the two-input Exclusive-OR, the sequential controllabilities are:

$$SC^0(Y) = \text{Min}\{SC^0(X_1) + SC^0(X_2), SC^1(X_1) + SC^1(X_2)\}$$
$$\text{SC}^1(\text{Y}) = \text{Min}\{SC^0(X_1) + SC^1(X_2), SC^1(X_1) + SC^0(X_2)\}$$ ■■

When computing sequential controllabilities through combinational logic, the value is not incremented. The intent of a sequential controllability number is to provide an estimate of the number of time frames needed to provide a 0 or 1 at a given node. Propagation through combinational logic does not affect the number of time frames.

When deriving equations for sequential circuits, both combinational and sequential controllabilities are computed, but the roles are reversed. The sequential controllability is incremented by 1, but an increment is not included in the combinational controllability equation. The creation of equations for a sequential circuit will be illustrated by means of an example.

Example Consider a positive edge triggered flip-flop with an active low reset but without a set capability. Then, 0-controllability is computed with

$$CC^0(Q) = \text{Min}\{CC^0(R), CC^1(R) + CC^0(D) + CC^0(C) + CC^1(C)\}$$
$$\text{SC}^0(Q) = \text{Min}\{SC^0(R), SC^1(R) + SC^0(D) + SC^0(C) + SC^1(C)\} + 1$$

and 1-controllability is computed with

$$CC^1(Q) = CC^1(R) + CC^1(D) + CC^0(C) + CC^1(C)$$
$$SC^1(Q) = SC^1(R) + SC^1(D) + SC^0(C) + SC^1(C) + 1$$ ■■

The first two equations state that a 0 can be obtained on the output of the delay flip-flop in either of two ways. It can be obtained either by setting the reset line to 0, or it can be obtained by setting the reset line to 1, setting the data line to 0, and then creating a rising edge on the clock line. Since four events must occur in the second choice, the controllability figure is the sum of the controllabilities of the four events. The sequential equation is incremented by 1 to reflect the fact that an additional time image is required to propagate a signal through the flip-flop. (This is not strictly true since a reset will produce a 0 at the Q output in the same time frame.) A 1 can be achieved only by clocking a 1 through the data line and that also requires holding the reset line at a 1.

The Observability Equations The observability of a node is a function of both the observability and the controllability of other nodes. This can be seen in Figure 8.8. In order to observe the value at node P, it must be possible to observe the

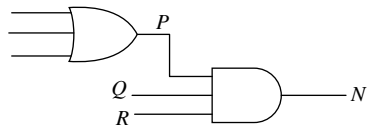


Figure 8.8 Node observability.

value on node *N*. If the value on node *N* cannot be observed at the output of the circuit and if node *P* has no other fanout, then clearly node *P* cannot be observed. However, to observe node *P* it is also necessary to place nodes *Q* and *R* into the 1 state. Therefore, a measure of the difficulty of observing node P can be computed with the following equation:

$$CO(P) = CO(N) + CC^1(Q) + CC^1(R) + 1$$

In general, the combinational observability of the output of a logic gate that drives the input of an AND gate is equal to the observability of that AND gate input, which in turn is equal to the sum of the observability of the AND gate output plus the 1-controllabilities of its other inputs, incremented by 1.

For a more general primitive combinational function, the observability of a given input can be computed from its propagation D-cubes (see Section 4.3.3). The process is as follows:

1. Select those D-cubes that have a D or $\overline{\text{D}}$ only on the input in question and 0, 1, or X on all the other inputs.
2. For each cube, add the 0- and 1-controllabilities corresponding to each input that has a 0 or 1 assigned.
3. Select the minimum controllability number computed over all the D-cubes chosen and add to it the observability of the output.

Example Given an AND-OR-Invert described by the equation $F = (\overline{\text{A} \cdot \text{B} + \text{C} \cdot \text{D}})$, the propagation D-cubes for input *A* are (D, 1, 0, X) and (D, 1, X, 0). The combinational observability for input *A* is equal to

$$CO(A) = \text{Min}\{CO(Z) + CC^1(B) + CC^0(C), CO(Z) + CC^1(B) + CC^0(D)\} + 1 \quad \blacksquare\blacksquare$$

The sequential observability equations, like the sequential controllability equations, are not incremented by 1 when computed through a combinational circuit. In general, the sequential controllability/observability equations are incremented by 1 when computed through a sequential circuit, but the corresponding combinational equations are not incremented.

Example Observability equations will be developed for the Reset and Clock lines of the delay flip-flop considered earlier. First consider the Reset line. Its observability can be computed using the following equations:

$$CO(R) = CO(Q) + CC^1(Q) + CC^0(R)$$
$$SO(R) = SO(Q) + SC^1(Q) + SC^0(R) + 1$$

Observability equations for the clock are as follows:

$$\begin{aligned} CO(C) = \text{Min}\{&CO(Q) + CC^1(Q) + CC^1(R) + CC^0(D) + CC^0(C) + CC^1(C), \\ &CO(Q) + CC^0(Q) + CC^1(R) + CC^1(D) + CC^0(C) + CC^1(C)\} \\ SO(C) = \text{Min}\{&SO(Q) + CC^1(Q) + SC^1(R) + SC^0(D) + SC^0(C) + SC^1(C), \\ &SO(Q) + SC^0(Q) + SC^1(R) + SC^1(D) + SC^0(C) + SC^1(C)\} + 1 \end{aligned}$$ ■■

Equations for the Reset line of the flip-flop assert that observability is equal to the sum of the observability of the Q output, plus the controllability of the flip-flop to a 1, plus the controllability of the Reset line to a 0. Expressed another way, the ability to observe a value on the Reset line depends on the ability to observe the output of the flip-flop, plus the ability to drive the flip-flop into the 1 state and then reset it. Observability of the clock line is described similarly.

The Algorithm Since the equations for the observability of an input to a logic gate or function depend on the controllabilities of the other inputs, it is necessary to first compute the controllabilities. The first step is to assign initial values to all primary inputs, I, and internal nodes, N:

$$\begin{aligned} CC^0(I) &= CC^1(I) = 1 \\ CC^0(N) &= CC^1(N) = \infty \\ SC^0(I) &= SC^1(I) = 1 \\ SC^0(N) &= SC^1(N) = \infty \end{aligned}$$

Having established initial values, each internal node can be selected in turn and the controllability numbers computed for that node, working from primary inputs to primary outputs, and using the controllability equations developed for the primitives. The process is repeated until, finally, the calculations stabilize. Node values must eventually converge since controllability numbers are monotonically nonincreasing integers.

Example The controllability numbers will be computed for the circuit of Figure 8.9. The first step is to initially assign a controllability of 1 to all inputs and ∞

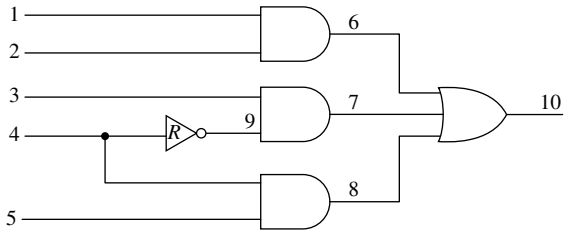


Figure 8.9 Controllability computations.

to all internal nodes. After the first iteration the 0- and 1-controllabilities of the internal nodes, in tabular form, are as follows:

N	$CC^0(N)$	$CC^1(N)$	$SC^0(N)$	$SC^1(N)$
6	2	3	0	0
7	2	∞	0	∞
8	2	3	0	0
9	2	2	0	0
10	7	4	0	0

After a second iteration the combinational 1-controllability of node 7 goes to a 4 and the sequential controllability goes to 0. If the nodes had been rank-ordered—that is, numbered according to the rule that no node is numbered until all its inputs are numbered—the second iteration would have been unnecessary. ■■

With the controllability numbers established, it is now possible to compute the observability numbers. The first step is to initialize all of the primary outputs, Y, and internal nodes, N, with

$$CO(Y) = 0$$
$$SO(Y) = 0$$
$$CO(N) = \infty$$
$$SO(N) = \infty$$

Then select each node in turn and compute the observability of that node. Continue until the numbers converge to stable values. As with the controllability numbers, observability numbers must eventually converge. They will usually converge much more quickly, with the fewest number of iterations, if nodes closest to the outputs are selected first and those closest to the inputs are selected last.

Example The observability numbers will now be computed for the circuit of Figure 8.9. After the first iteration the following table is obtained:

N	CO(N)	SO(N)
9	∞	∞
8	5	0
7	5	0
6	5	0
5	7	0
4	7	0
3	8	0
2	7	0
1	7	0

On the second iteration the combinational and sequential observabilities of node 9 settle at 7 and 0, respectively. ■■

SCOAP can be generalized using the D-algorithm notation (cf. Section 4.3.1). This will be illustrated using the truth table for the arbitrary function defined in Figure 8.10. In practice, this might be a frequently used primitive in a library of macrocells. The first step is to define the sets P_1 and P_0. Then create the intersection $P_1 \cap P_0$ and use the resulting intersections, along with the truth table, to create controllability and observability equations. The sets P_1 and P_0 are as follows:

$$P_1 = \{(0,0,0), (0,1,0), (1,0,1), (1,1,0)\} = \{(0,x,0), (1,0,1), (x,1,0)\}$$

$$P_0 = \{(0,0,1), (0,1,1), (1,0,0), (1,1,1)\} = \{(0,x,1), (1,0,0), (x,1,1)\}$$

The intersection table $P_1 \cap P_0$ is as follows:

A	*B*	*C*	*Z*
~~0~~	~~0~~	~~$\overline{D}$~~	~~D~~
$\overline{D}$	0	0	D
~~0~~	~~1~~	~~$\overline{D}$~~	~~D~~
D	0	1	D
1	0	D	D
1	$\overline{D}$	1	D
1	D	0	D
~~1~~	~~1~~	~~$\overline{D}$~~	~~D~~
1	*x*	$\overline{D}$	D
x	1	$\overline{D}$	D

A	*B*	*C*	*Z*
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

Figure 8.10 Truth table for arbitrary function.

Note first that some members of P_1 and P_0 were left out of the intersection table. The rows that were omitted were those that had either two or three D and/or $\overline{D}$ signals as inputs. This follows from the fact that SCOAP does not compute observability through multiple inputs to a function. Note also that three rows were crossed out and two additional rows were added at the bottom of the intersection table. The first of these added rows resulted from the intersection of rows 1 and 3. In words, it states that if input *A* is a 1, then the value at input *C* is observable at *Z* regardless of the value on input *B*. The second added row results from the intersection of rows 3 and 8. The following controllability and observability equations for this function are derived from P_0, P_1, and their intersection:

$$CO(A) = \min\{CC^0(B) + CC^0(C), CC^0(B) + CC^1(C)\} + CO(Z) + 1$$

$$CO(B) = \min\{CC^1(A) + CC^1(C), CC^1(A) + CC^0(C)\} + CO(Z) + 1$$

$$CO(A) = \min\{CC^0(A), CC^1(A) + CC^0(B), CC^1(B)\} + CO(Z) + 1$$

$$CC^0(Z) = \min\{CC^0(A) + CC^1(C), CC^1(A) + CC^0(B) + CC^0(C), CC^1(B) + CC^1(C)\} + 1$$

$$CC^1(Z) = \min\{CC^0(A) + CC^0(C), CC^1(A) + CC^0(B) + CC^1(C), CC^1(B) + CC^0(C)\} + 1$$

8.3.2 Other Testability Measures

Other algorithms exist, similar to SCOAP, which place different emphasis on circuit parameters. COP (controllability and observability program) computes controllability numbers based on the number of inputs that must be controlled in order to establish a value at a node.[3] The numbers therefore do not reflect the number of levels of logic between the node being processed and the primary inputs. The SCOAP numbers, which encompass both the number of levels of logic and the number of primary inputs affecting the C/O numbers for a node, are likely to give a more accurate estimate of the amount of work that an ATPG must perform. However, the number of primary inputs affecting C/O numbers perhaps reflects more

accurately the probability that a node will be switched to some value randomly; hence it may be that it more closely correlates with the probability of random fault coverage when simulating test vectors.

Testability analysis has been extended to functional level primitives. FUNTAP (functional testability analysis program)[4] takes advantage of structures such as n-wide data paths. Whereas the single net may have binary values 0 and 1, and these values can have different C/O numbers, the n-wide data path made up of binary signals may have a value ranging from 0 to $2^n - 1$. In FUNTAP no significance is attached to these values; it is assumed that the data path can be set to any value i, $0 \le i \le 2^n - 1$, with equal ease or difficulty. Therefore, a single controllability number and a single observability number are assigned to all nets in a data path, independent of the logic values assigned to individual nets that make up the data path.

The ITTAP program[5] computes controllability and observability numbers, but, in addition, it computes parameters TL0, TL1, and TLOBS, which measure the length of the sequence needed in sequential logic to set a net to 0 or 1 or to observe the value on that node. For example, if a delay flip-flop has a reset that can be used to reset the flip-flop to 0, but can only get a 1 by clocking it in from the Data input, then TL0 = 1 and TL1 = 2.

A more significant feature of ITTAP is its selective trace capability. This feature is based on two observations. First, controllabilities must be computed before observabilities, and second, if the numbers were once computed, and if a change is made to enhance testability, numbers need only be recomputed for those nodes where the numbers can change. The selection of elements for recomputation is similar to event-driven simulation. If the controllability of a node changes because of the addition of a test point, then elements driven by that element must have their controllabilities recomputed. This continues until primary outputs are reached or elements are reached where the controllability numbers at the outputs are unaffected by changing numbers at the inputs. At that point, the observabilities are computed back toward the inputs for those elements with changed controllability numbers on their inputs.

The use of selective trace provides a savings in CPU time of 90–98% compared to the time required to recompute all numbers in a given circuit. This makes it ideal for use in an interactive environment. The designer visually inspects either a circuit or a list of nodes at a video display terminal and then assigns a test point and immediately views the results. Because of the quick response, the test point can be shifted to other nodes and the numbers recomputed. After several such iterations, the logic designer can settle on the node that provides the greatest improvement in the C/O numbers.

The interactive strategy has pedagogical value. Placing a test point at a node with the worst C/O numbers is not always the best solution. It may be more effective to place a test point at a node that controls the node in question, since this may improve controllability of several nodes. Also, since observability is a function of controllability, greatest improvements in testability may sometimes be had by assigning a test point as an input to a gate rather than as an output, even though the analysis program indicates that the observability is poor. The engineer who uses the interactive tool,

particularly recent graduates who may not have given much thought to testability issues, may learn with such an interactive tool how best to design for testability.

8.3.3 Test Measure Effectiveness

Studies have been conducted to determine the effectiveness of testability analysis. Consider the circuit defined by the equation

$$F = A \cdot (B + C + D)$$

An implementation can be realized by a two-input AND gate and a three-input OR gate. With four inputs, there are 16 possible combinations on the inputs. An SA1 fault on input A to the AND gate has a 7/16 probability of detection, whereas an SA0 on any input to the OR gate has a 1/16 probability of detection. Hence a randomly generated 4-bit vector applied to the inputs of the circuit is seven times as likely to detect the fault on the AND gate input as it is to detect a fault on a particular OR gate input.

Suppose controllability of a fault is defined as the fraction of input vectors that set a faulty net to a value opposite its stuck-at value, and observability is defined as the fraction of input vectors that propagate the fault effect to an output.[6] Testability is then defined as the fraction of input vectors that test the fault. Obviously, to test a fault, it is necessary to both control and observe the fault effect; hence testability for a given fault can be viewed as the number of vectors in the intersection of the controllability and observability sets, divided by the total number of vectors. But, there may be two reasonably large sets whose intersection is empty. A simple example is shown in Figure 8.11. The controllability for the bottom input of gate numbered 1 is 1/2. The observability is 1/4. Yet, the SA1 on the input cannot be detected because it is redundant.

In another investigation of testability measures, the authors attempt to determine a relationship between testability figures and detectability of a fault.[7] They partitioned faults into classes based on testability estimates for the faults and then plotted curves of fault coverage versus vector number for each of these classes. The curves were reasonably well behaved, the fault coverage curves rising more slowly, in general, for the more difficult to test fault classes, although occasionally a curve for some particular class would rise more rapidly than the curve for a supposedly easier to test class of faults. They concluded that testability data were a poor predictor of fault detection for individual faults but that general information at the circuit level was available and useful. Furthermore, if some percentage, say 70%, of a class of difficult to test faults are tested, then any fixes made to the circuit for testability purposes have only a 30% chance of being effective.

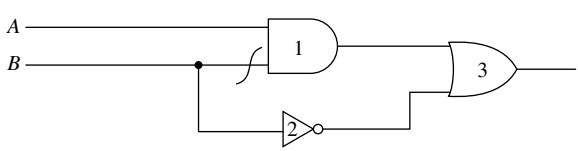


Figure 8.11 An undetectable fault.

8.3.4 Using the Test Pattern Generator

If test vectors for a circuit are to be generated by an ATPG, then the most direct way in which to determine its testability is to simply run the ATPG on the circuit. The ability (or inability) of an ATPG to generate tests for all or part of a design is the best criterion for testability. Furthermore, it is a good practice to run test pattern generation on a design before the circuit has been fabricated. After a board or IC has been fabricated, the cost of incorporating changes to improve testability increases dramatically.

A technique employed by at least one commercial ATPG employs a preprocess mode in which it attempts to set latches and flip-flops to both the 0 and 1 state before attempting to create tests for specific faults in a circuit.[8] The objective is to find troublesome circuits before going into test pattern generation mode. The ATPG compiles a list of those flip-flops for which it could not establish the 0 and/or 1 state. Whenever possible, it indicates the reason for the failure to establish desired value(s). The failure may result from such things as races in which relative timing of the signals is too close to call with confidence, or it could be caused by bus conflicts resulting from inability to set one or more tri-state control lines to a desired value. It could also be the case that controllability to 0 or 1 of a flip-flop depends on the value of another flip-flop that could not be controlled to a critical value. It also has criteria for determining whether the establishment of a 0 or 1 state took an excessive amount of time.

Analysis of information in the preprocess mode may reveal clusters of nodes that are all affected by a single uncontrollable node. It is also important to bear in mind that nodes which require a great deal of time to initialize can be as detrimental to testability as nodes that cannot be initialized. An ATPG may set arbitrary limits on the amount of time to be expended in trying to set up a test for a particular fault. When that threshold is exceeded, the ATPG will give up on the fault even though a test may exist.

C/O numbers can be used by the ATPG to influence the decision-making process. On average, this can significantly reduce the amount of time required to create test patterns. The C/O numbers can be attached to the nodes in the circuit model, or the numbers can be used to rearrange the connectivity tables used by the ATPG, so that the ATPG always tries to propagate or justify the easiest to control or observe signals first. Initially, when a circuit model is read into the ATPG, connectivity tables are constructed reflecting the interconnections between the various elements in the circuit. A FROM table lists the inputs to an element, and a TO table lists the elements driven by a particular element.

By reading observability information, the ATPG can sort the elements in the TO table so that the most observable path is selected first when propagating elements. Likewise, when justifying logic values, controllability information can be used to select the most controllable input to the gate. For example, when processing an AND gate, if it is necessary to justify a 0 on the output of the AND gate, then the input with the lowest 0-controllability should be tried first. If it cannot be justified, then attempt the other inputs, always selecting as the next choice the input, not yet attempted, that is judged to be most controllable.

8.4 THE SCAN PATH

Ad hoc DFT methods can be useful in small circuits that have high yield, as well as circuits with low sequential complexity. For ICs on small die with low gate count, it may be necessary to get only a small boost in fault coverage in order to achieve required AQL, and one or more ad hoc DFT solutions may be adequate. However, a growing number of design starts are in the multi-million transistor range. Even if it were possible to create a test with high-fault coverage, it would in all likelihood take an unacceptably long time on a tester to apply the test to an IC. However, it is seldom the case that an adequate test can be created for extremely complex devices using traditional methods. In addition to the length of the test, test development cost continues to grow. Another factor of growing importance is customer expectations. As digital products become more pervasive, they increasingly are purchased by customers unsympathetic to the difficulties of testing, they just want the product to work. Hence, it is becoming imperative that devices be free of defects when shipped to customers.

The aforementioned factors increase the pressure on vendors to produce fault-free products. The ever-shrinking feature sizes of ICs simultaneously present both a problem and an opportunity for vendors. The shrinking feature sizes make the die susceptible to defects that might not have affected it in a previous generation of technology. On the other hand, it affords an opportunity to incorporate more test related features on the die. Where die were once core-limited, now the die are more likely to be pad-limited (cf. Figure 8.12). In core-limited die there may not be sufficient real estate on the die for all the features desired by marketing; as a result, testability was often the first casualty in the battle for die real estate. With pad-limited die, larger and more complex circuits, and growing test costs, the argument for more die real estate dedicated to test is easier to sell to management.

8.4.1 Overview

Before examining scan test, consider briefly the circuit of Problem 8.10, an eight-state sequential circuit implemented as a muxed state machine. It is fairly easy to generate a complete test for the circuit because it is a completely specified state machine (CSSM); that is, every state defined by the flip-flops can be reached from some other state in one or more transitions. Nonetheless, generating a test program

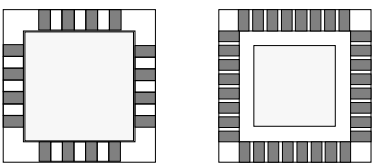


Figure 8.12 The changing face of IC design.

becomes quite tedious because of all the details that must be maintained while propagating and justifying logic assignments through the time and logic dimensions. The task becomes orders of magnitude more difficult when the state machine is implemented using one-hot encoding. In that design style, every state is represented by a unique flip-flop, and the circuit becomes an incompletely specified state machine (ISSM)—that is, one in which n flip-flops implement n legal states out of 2^n possible states. Backtracing and justifying logic values in the circuit becomes virtually impossible.

Regardless of how the circuit is implemented, with three or eight flip-flops, the test generation task for a fault in combinational logic becomes much easier if it were possible to compute the required test values at the I/O pins and flip-flops, and then load the required values directly into the flip-flops without requiring several vectors to transition to the desired state. The scan path serves this purpose. In this approach the flip-flops are designed to operate either in parallel load or serial shift mode. In operational mode the flip-flops are configured for parallel load. During test the flip-flops are configured for serial shift mode. In serial shift mode, logic values are loaded by serially shifting in the desired values. In similar fashion, any values present in the flip-flops can be observed by serially clocking out their contents.

A simple means for creating the scan path consists of placing a multiplexer just ahead of each flip-flop as illustrated in Figure 8.13. One input to the 2-to-1 multiplexer is driven by normal operational data while the other input—with one exception—is driven by the output of another flip-flop. At one of the multiplexers the serial input is connected to a primary input pin. Likewise, one of the flip-flop outputs is connected to a primary output pin. The multiplexer control line, also connected to a primary input pin, is now a mode control; it can permit parallel load for normal operation or it can select serial shift in order to enter scan mode. When scan mode is selected, there is a complete serial shift path from an input pin to an output pin.

Since it is possible to load arbitrary values into flip-flops and read the contents directly out through the serial shift path, ATPG requirements are enormously simplified. The payoff is that the complexity of testing is significantly reduced because it is no longer necessary to propagate tests through the time dimension represented by sequential circuits. The scan path can be tested by shifting a special pattern through

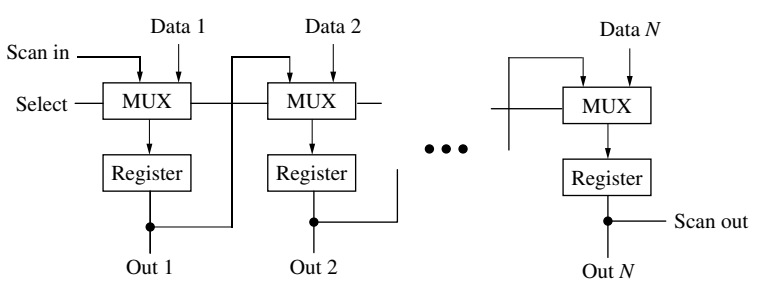


Figure 8.13 A scan path.

the scan path before even beginning to address stuck-at faults in the combinational logic. A test pattern consisting of alternating pairs of 1s and 0s (i.e., 11001100....) will test the ability of the scan path to shift all possible transitions. This makes it possible for the ATPG to ignore faults inside the flip-flops, as well as stuck-at faults on the clock circuits.

During the generation of test patterns, the ATPG treats the flip-flops as I/O pins. A flip-flop output appears to be a combinational logic input, whereas a flip-flop input appears to be a combinational logic output. When an ATPG is propagating a sensitized path, it stops at a flip-flop input just as it would stop at a primary output. When justifying logic assignments, the ATPG stops at the output of flip-flops just as it would stop at primary inputs. The only difference between the actual I/O pins and flip-flop "I/O pins" is the fact that values on the flip-flops must be serially shifted in when used as inputs and serially shifted out when used as outputs.

When a circuit with scan path is used in its normal mode, the mode control, or test control, is set for parallel load. The multiplexer selects normal operational data and, except for the delay through the multiplexer, the scan circuitry is transparent. When the device is being tested, the mode control alternates between parallel load and serial shift. This is illustrated in Figure 8.14.

The figure assumes a circuit composed of four scan-flops that, during normal mode, are controlled by positive clock edges. Data are serially shifted into the scan path when the scan-enable is high. After all of the scan-flops are loaded, the scan-enable goes low. At this point the next clock pulse causes normal circuit operation using the data that were serially shifted into the scan-flops. That data pass through the combinational logic and produce a response that is clocked into destination scan-flops. Note that data present at the scan-input are ignored during this clock period. After one functional clock has been applied, scan-enable again becomes active. Now the Clk signal again loads the scan-flops. During this operation, response data are also captured at the scan-out pin. That data are compared to expected data to determine whether or not any faults are present in the circuit.

The use of scan tremendously simplifies the task of creating test stimuli for sequential circuits, since the circuit is essentially reduced to a combinational ATPG for test purposes, and algorithms for those circuits are well understood, as we saw in Chapter 4. It is possible to achieve very high fault coverage, often in the range of

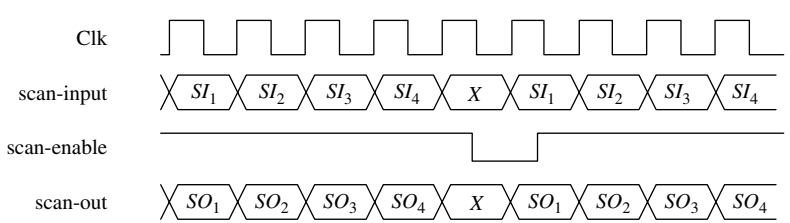


Figure 8.14 Scan shift operation.

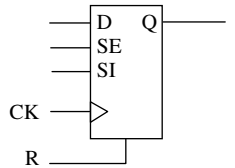


Figure 8.15 Scan flip-flop symbol.

97–99% for the parts of the circuit that can be tested with scan. Equally important for management, the amount of time required to generate the test patterns and achieve a target fault coverage is predictable. Scan can also help to reduce time on the tester since, as we shall see, multiple scan paths can run in parallel. However, it does impose a cost. The multiplexers and the additional metal runs needed to connect the mode select to the flip-flops can require from 5% to 20% of the real estate on an IC. The performance delay introduced by the multiplexers in front of the flip-flops may impose a penalty of from 5% to 10%, depending on the depth of the logic.

8.4.2 Types of Scan-Flops

The simplest form of scan-flop incorporates a multiplexer into a macrocell together with a delay flip-flop. A common symbol denoting a scan-flop is illustrated in Figure 8.15. Operational data enter at *D*, while scan data enter at *SI*. The scan enable, *SE*, determines which data are selected and clocked into the flip-flop.

Dual Clock Serial Scan An implementation of scan with dual clocks is shown in Figure 8.16.[9] In this implementation, comprised of CMOS transmission gates, the goal was to have the least possible impact on circuit performance and area overhead.

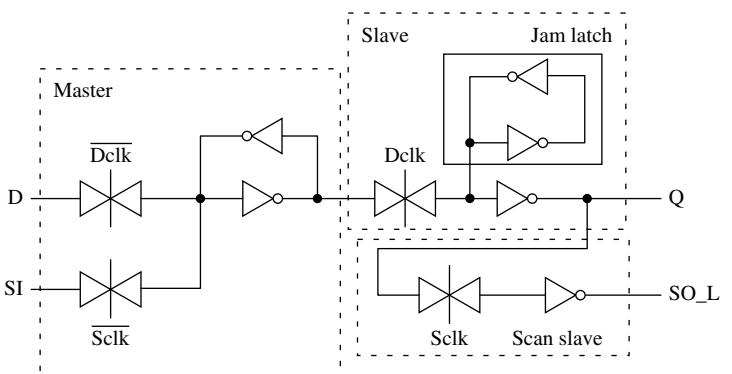


Figure 8.16 Flip-flop with dual clock.

Dclk is used in operational mode, and Sclk is the scan clock. Operational data and scan data are multiplexed using $\overline{\text{Dclk}}$ and $\overline{\text{Sclk}}$. When operating in scan mode, Dclk is held high and Sclk goes low to permit scan data to pass into the Master latch. Because Dclk is high, the scan data pass through the Slave latch and, when Sclk goes high, pass through the Scan slave and appears at SO_L.

Addressable Registers Improved controllability and observability of sequential elements can be obtained through the use of addressable registers.[10] Although, strictly speaking, not a scan or serial shift operation, the intent is the same—that is, to gain access and control of sequential storage elements in a circuit. This approach uses *X* and *Y* address lines, as illustrated in Figure 8.17. Each latch has an *X* and *Y* address, as well as clear and preset inputs, in addition to the usual clock and data lines. A scan address goes to *X* and *Y* decoders for the purpose of generating the *X* and *Y* signals that select a latch to be loaded. A latch is forced to a 1 (0) by setting the address lines and then pulsing the Preset (Clear) line.

Readout of data is also accomplished by means of the *X* and *Y* addresses. The selected element is gated to the SDO (Serial Data Out) pin, where it can be observed. If there are more address lines decoded than are necessary to observe latches, the extra *X* and *Y* addresses can be used to observe nodes in combinational logic. The node to be observed is input to a NAND gate along with *X* and *Y* signals, as a latch would be; when selected, its value appears at the SDO.

The addressable latches require just a few gates for each storage element. Their affect on operation during normal operation is negligible, due mainly to loading caused by the NAND gate attached to the *Q* output. The scan address could require several I/O pins, but it could also be generated internally by a counter that is initially reset and then clocked through consecutive addresses to permit loading or reading of the latches.

Random access scan is attractive because of its negligible effect on IC performance and real estate. It was developed by a mainframe company where performance, rather than die area, was the overriding issue. Note, however, that with shrinking component size the amount of area taken by interconnections inside an IC grows more significant; the interconnect represents a larger percentage of total chip

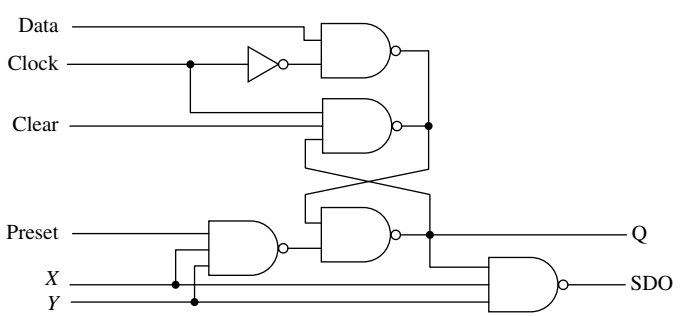


Figure 8.17 Addressable flip-flop.

area. The addressable latches require that several signal lines be routed to each addressable latch, and the chip area occupied by these signal lines becomes a major factor when assessing the cost versus benefits of the various methods.

8.4.3 Level-Sensitive Scan Design

Much of what is published about DFT techniques is not new. They have been described as early as December 1963[11] and again in April 1964.[12] Detailed description of a scan path and its proposed use for testability and operational modes is described in a patent filed in 1968.[13] Discussion of scan path and derivation of a formal cost model were published in 1973.[14] The level-sensitive scan design (LSSD) methodology was introduced in a series of papers presented at the Design Automation Conference in 1977.[15–17]

LSSD extends DFT beyond the scan concept. It augments the scan path with additional rules whose purpose is to cause a design to become level sensitive. A *level-sensitive* system is one in which the steady-state response to any allowed input state change is independent of circuit and wire delays within the system. In addition, if an input state change affects more than one input signal, then the response must be independent of the order in which they change.[15] The object of these rules is to preclude the creation of designs in which correct operation depends on critical timing factors.

To achieve this objective, the memory devices used in the design are level-sensitive latches. These latches permit a change of internal state at any time when the clock is in one state, usually the high state, and inhibit state changes when the clock is in the opposite state. Unlike edge-sensitive flip-flops, the latches are insensitive to rising and falling edges of pulses, and therefore the designer cannot create circuits in which correct operation depends on pulses that are themselves critically dependent on circuit delay. The only timing that must be taken into account is the total propagation time through combinational logic between the latches.

In the LSSD environment, latches are used in pairs as illustrated in Figure 8.18. These latch pairs are called shift-register latches (SRL), and their operation is controlled by multiple clocks, denoted *A*, *B*, and *C*. The *Data* input is used in operational mode whereas *Scan-in*, which is driven by the *L2* output of another SRL, is used in the scan mode. During operational mode the *A* clock is inactive. The *C* clock is used to clock data into *L1* from the *Data* input, and output can be taken from either *L1* or *L2*. If output is taken from *L2,* then two clock signals are required. The second signal, called the *B* clock, clocks data into *L2* from the *L1* latch. This configuration is sometimes referred to as a *double latch* design.

When the scan path is used for testing purposes, the *A* clock is used in conjunction with the *B* clock. Since the *A* clock causes data at the Scan-in input to be latched into L1, and the Scan-in signal comes from the L2 output of another SRL (or a primary input pin), alternately switching the *A* and *B* clocks serially shifts data through the scan path from the Scan-in terminal to the Scan-out terminal.

Conceptually, LSSD behaves much like the dual-clock configuration discussed earlier. However, there is more to LSSD, namely, a set of rules governing the manner in which logic is clocked. Consider the circuit depicted in Figure 8.19. If S1, S2,

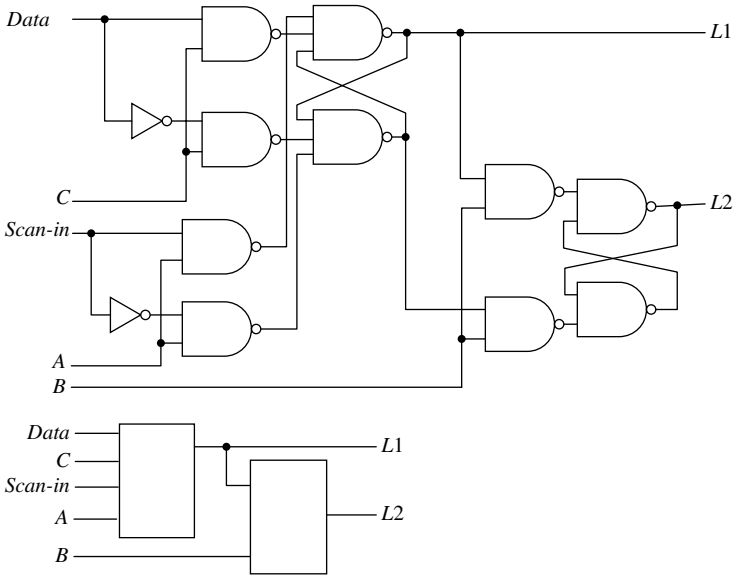


Figure 8.18 The shift register latch.

and S3 are L1 latches, the correct operation of the circuit depends on relative timing between the clock and data signals. When the clock is high, there is a direct combinational logic path from the input of S1 to the output of S3. Since the clock signal must stay high for some minimum period of time in order to latch the data, this direct combinational path will exist for that duration.

In addition, the signal from S1 to S2 may go through a very short propagation path. If the clock does not drop in time, input data to the S1 latch may not only get

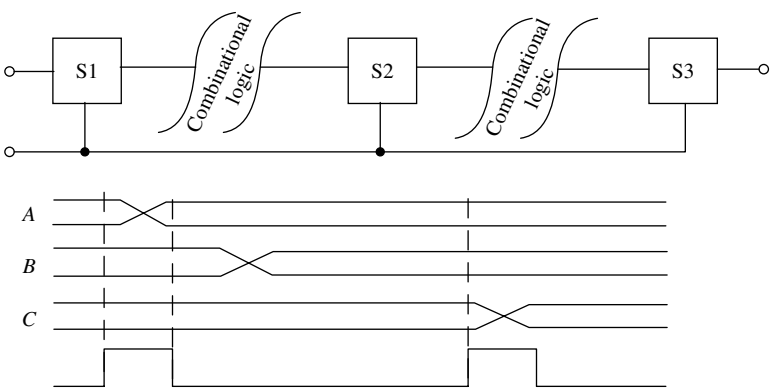


Figure 8.19 Some timing problems.

latched in S1 but may reach S2 and get latched into S2 a clock period earlier than intended. Hence, as illustrated in waveform *A* the short propagation path can cause unpredictable results. Waveform *C* illustrates the opposite problem. The next clock pulse appears before new data reaches S2. Clearly, for correct behavior it is necessary that the clock cycle be as short as possible, but it must not be shorter than the propagation time through combinational logic.

The use of the double latch design can eliminate the situation in waveform *A*. To resolve this problem, LSSD imposes restrictions on the clocking of latches. The rules will be listed and then their effect on the circuit of Figure 8.19 will be discussed.

1. Latches are controlled by two or more nonoverlapping clocks such that a latch *X* may feed the data port of another latch *Y* if and only if the clock that sets the data into latch *Y* does not clock latch *X*.
2. A latch *X* may gate a clock C_1 to produce a gated clock C_2 that drives another latch *Y* if and only if clock C_3 does not clock latch *X*, where C_3 is any clock produced from C_1.
3. It must be possible to identify a set of clock primary inputs from which the clock inputs to SRLs are controlled either through simple powering trees or through logic that is gated by SRLs and/or nonclock primary inputs.
4. All clock inputs to all SRLs must be at their off states when all clock primary inputs are held to their off states.
5. The clock signal that appears at any clock input of an SRL must be controlled from one or more clock primary inputs such that it is possible to set the clock input of the SRL to an on state by turning any one of the corresponding primary inputs to its on state and also setting the required gating condition from SRLs and/or nonclock primary inputs.
6. No clock can be ANDed with the true value or complement value of another clock.
7. Clock primary inputs may not feed the data inputs to latches, either directly or through combinational logic, but may only feed the clock input to the latches or the primary outputs.

Rule 1 forbids the configuration shown in Figure 8.19. A simply way to comply with the rules is to use both the L1 and L2 latches and control them with nonoverlapping clocks as shown in Figure 8.20. Then the situation illustrated in waveform *A* will not occur. The contents of the L2 latch cannot change in response to new data at its input as long as the *B* clock remains low. Therefore, the new data entering the L1 latch of SRL S1, as a result of clock *C* being high, cannot get through its L2 latch, because the *B* clock is low and hence cannot reach the input of SRL S2. The input to S2 remains stable and is latched by the *C* clock.

The use of nonoverlapping clocks will protect a design from problems caused by short propagation paths. However, the time between the fall of clock *C* and the rise

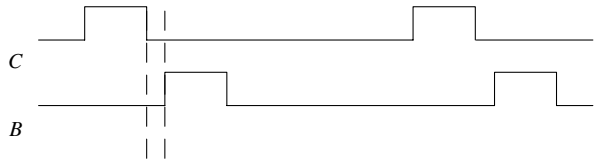


Figure 8.20 The two-clock signal.

of clock *B* is "dead time"; that is, once the data are latched into L1, the goal is to move it into L2 as quickly as possible in order to realize maximum performance. Thus, the interval from the fall of *C* to the rise of *B* in Figure 8.20 should be as brief as possible without, however, making the duration too short. In a chip with a great many wire paths, the two clocks may be nonoverlapping at the I/O pins and yet may overlap at one or more SRLs inside the chip due to signal path delays. This condition is referred to as *clock skew*. When debugging a design, experimentation with clock edge separation can help to determine whether clock skew is causing problems. If clock skew problems exist, it may be necessary to change the layout of a chip or board, or it may require a greater separation of clock edges to resolve the problem.

The designer must still be concerned with the configuration in waveform C; that is, the clock cycle must exceed the propagation delay of the longest propagation path. However, it is a relatively straightforward task to compute propagation delays along combinational logic paths. Timing verification, as described in Section 2.13, can be used to compute the delay along each path and then print out all critical paths that exceed a specified threshold. The design team can elect to redesign the critical paths or increase the clock cycle.

Test program development using the LSSD scan path closely follows the technique used with other scan paths. One interesting variant when testing is the fact that the scan path itself can be checked with what is called a flush test.[16] In a flush test the *A* and *B* clocks are both set high. This creates a direct combinational path from the scan-in to the scan-out. It is then possible to apply a logic 1 and 0 to the scan-in and observe them directly at the scan output without further exercising the clocks. This flush test exercises a significant portion of the scan path. The flush test is followed by clocking 1s and 0s through the scan path to ensure that the clock lines are fault-free.

Another significant feature of LSSD, as implemented, is the fact that it is supported by a design automation system that enforces the design rules.[17] Since the design automation system incorporates much knowledge of LSSD, it is possible to check the design for compliance with design rules. Violations detected by the checking programs can be corrected before the design is fabricated, thus ensuring that design violations will not compromise the testability goals that were the object of the LSSD rules.

The other DFT approaches discussed, including non-LSSD scan and addressable registers, do not, in and of themselves, inhibit some design practices that traditionally

have caused problems for ATPGs. They require design discipline imposed either by the logic designers or by some designated testability supervisor. LSSD, by requiring that designs be entered into a design data base via design automation programs that can check for rule violations, makes it difficult to incorporate design violations without concurrence of the very people who are ultimately responsible for testing the design.

8.4.4 Scan Compliance

The intent of scan is to make a circuit testable by causing it to appear to be strictly combinational to an ATPG. However, not all circuits can be directly transformed into combinational circuits by adding a scan path. Consider the self-resetting flip-flop in Figure 8.21. Any attempt to serially shift data through the scan-in (SI) will be defeated by the self-resetting capability of flip-flop S_2. The self-resetting capability not only forces S_2 back to the 0 state, but the effect on S_3, as data are scanned through, is unpredictable. Whether or not scan data reach S_3 from S_2 will depend on the value of the Delay as well as the period of the clock.

A number of other circuit configurations create similar complications. This includes configurations such as asynchronous set and clear inputs and flip-flops whose clock, set, and/or clear inputs are driven by combinational logic. Two problems result when flip-flops are clocked by derived clocks—that is, clocks generated from subcircuits whose inputs are other clocks and random logic signals. The first of these problems is that an ATPG may have difficulty creating the clocking signal and keeping it in proper synchronization with clock signals on other flip-flops. The other problem is the fact that the derived clock may be glitchy due to races and hazards. So, although the circuit may work correctly during normal operation, test vectors generated by an ATPG may create input combinations not intended by the designers of the circuit and, as a result, the circuit experiences races and hazards that do not occur during normal operation.

Latches are forbidden by some commercial systems that support scan. Scan-based ATPG tools expect the circuit they are processing to be a pure combinational circuit. Since the latches hold state information, logic values emanating from the latches are unpredictable. Therefore, those values will be treated as Xs. This can cause a considerable amount of logic to become untestable. One way to implement

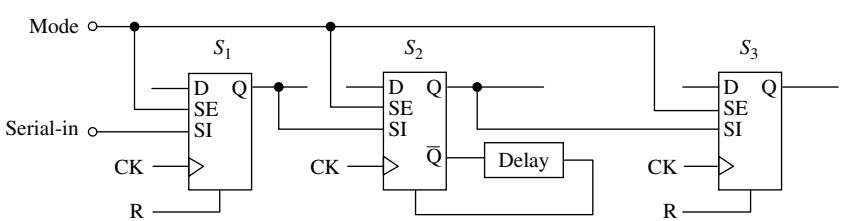


Figure 8.21 A reset problem.

testable latches is shown in Figure 8.22.[18] When in test mode, the *TestEnable* signal is held fixed at 1, thus blocking the feedback signals. As a result, the NAND gates appear, for purposes of test, to be inverters. A slight drawback is that some faults become undetectable. But this is preferable to propagating Xs throughout a large block of combinational logic.

If there are D latches present in the circuit—that is, those with Data and Enable inputs—then a *TestEnable* signal can be ORed with the Enable signal. The *TestEnable* signal can be held at logic 1 during test so that the D latch appears, for test purposes, to be a buffer or inverter.

Many scan violations can be resolved through the use of multiplexers. For example, if a circuit contains a combinational feedback loop, then a multiplexer can be used to break up the loop. This was illustrated in Figure 8.3 where the configuration was used to avoid gating the clock signal. To use this configuration for test, the Load signal selects the feedback loop during normal operation, but selects a test input signal during test. The test input can be driven by a flip-flop that is included in the scan chain but is dedicated to test, that is, the flip-flop is not used during normal operation. This circuit configuration may require two multiplexers; One is used to select between Load and Data, and the second one is used to choose between scan-in and normal operation.

Tri-state circuits can cause problems because they are often used when two or more devices are connected to a bus. When several drivers are connected to a bus, it is sometimes the case that none of the drivers are active, causing the bus to enter the unknown state. When that occurs, the X on the bus may spread throughout much of the logic, thus rendering a great deal of logic untestable for those vectors when the bus is unknown.

One way to prevent conflicts at buses with multiple drivers is to use multiplexers rather than tri-state drivers. Then, if there are no signals actively driving the bus, it can be made to default to either 0 or 1. If tri-state drivers are used, a 1-of-n selector can be used to control the tri-state devices. If the number of bus drivers n is $2^{d-1} < n < 2^d$, there will be combinations of the 2^d possible selections for which no signal is driving the bus. The unused combinations can be set to force 0s or 1s onto the bus. This is illustrated in Figure 8.23, where $d = 2$, and one of the four bus drivers is connected to ground. If select lines S_1 and S_2 do not choose any of D_1, D_2, or D_3, then the *Bus* gets a logic 0. Note that while the solution in Figure 8.23 maintains the bus at a known value regardless of the values of S_1 and S_2, a fault on a tri-state enable line can cause the faulty bus to assume an indeterminate value, resulting in at best a

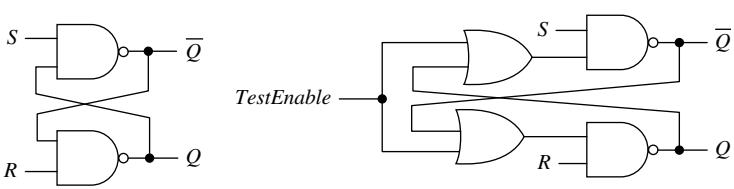


Figure 8.22 Testable NAND latch.

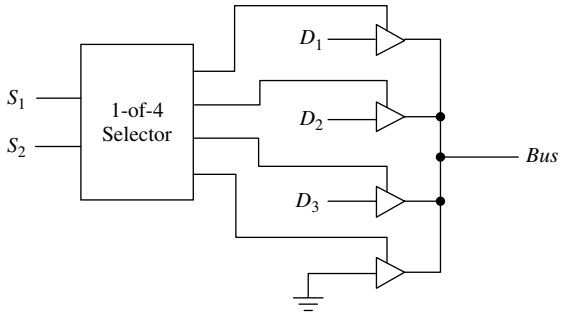


Figure 8.23 Forcing a bus to a known value.

probable detect. When a multiplexer is used, both good and faulty circuits will have known, but different, values.

A potentially more serious situation occurs if a circuit is designed in such a way that two or more drivers may be simultaneously active during scan test. For example, the tri-state enables may be driven, directly or indirectly, by flip-flops. If two or more drivers are caused to become active during scan and if they are attempting to drive the circuit to opposite values, the test can damage the very circuit it is attempting to evaluate for correct operation.

8.4.5 Scan-Testing Circuits with Memory

With shrinking feature sizes, increasing numbers of ICs are being designed with memory on the same die with random logic. Memory often takes up 80% or more of the transistors on a die in microprocessor designs while occupying less than half the die area (cf. Section 10.1). Combining memory and logic on a die has the advantages of improved performance and reliability. However, ATPG tools generally treat memory, and other circuitry such as analog circuits, as black boxes. So, for scan test, these circuits must be treated as exceptions. In the next two chapters we will deal with built-in self-test (BIST) for memories, here we will consider means for isolating or bypassing the memory so that the remainder of the IC can be tested.

The circuit in Figure 8.24 illustrates the presence of *shadow logic* between scan registers and memory.[19] This is combinational logic that can not be directly accessed by the scan circuits. If the shadow logic consists solely of addressing logic, then it is testable by BIST. However, if other random logic is present, it may be necessary to take steps to improve controllability and observability. Observability of signals at the address and data inputs can be accomplished by means of the observability tree in Figure 8.4. Controllability of logic between memory output and the scan register can be achieved by multiplexing the memory Data-out signals with scanned in test data.

An alternative is to multiplex the address and Data-in signals with the Data-out signals as shown in Figure 8.24. In test mode a combinational path exists from the input side of memory to the output side. Address and data inputs can be exclusive-OR'ed so that there are a total of n signals on both of the multiplexer input ports. For

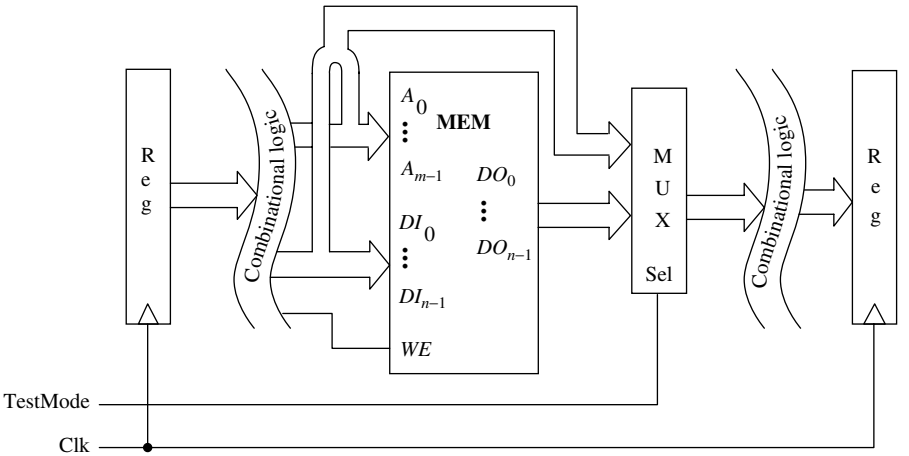


Figure 8.24 Memory with shadow logic.

example, if $m = 2n$, then A_{2i}, A_{2i+1}, and D_i can be exclusive-OR'ed, for $0 \le i < n$, to reduce the number of inputs to the multiplexer to n. Note that it may be necessary to inhibit memory control signals while performing the scan test.

It might be possible, for test generation purposes, to remodel a memory as a register, then force values on the memory control pins that cause the address lines to assume a fixed value, such as 0, during test. Better still, it might be possible to make the memory completely transparent. In the transparent memory test mode, with the right values on the control lines, Data-in flows directly to Data-out so that the memory appears, for test purposes, to be direct connections between Data-in and Data-out.

If the memory has a bidirectional Data port connected to a bus, the best approach may be to disable the memory completely while testing the random logic. This may require that the TestMode signal be used to disable the OE (output enable) during scan. Then if there is logic that is being driven by the bus, it may be necessary to substitute some other source for that test data. Perhaps it will be necessary to drive the bus from an input port during test.

Another method for dealing with memories is to write data into memory before scan tests are generated. Suppose the memory has an equal number of address and data inputs. Then, before running the scan test on the chip, run a test program that loads memory with all possible values. For example, if there are n address lines and n data lines, load location i with the value i, for $0 \le i < 2^n$. Then, during scan test the write enable is disabled. During test pattern generation the circuit is remodeled so that either the address or data inputs are connected directly to the data outputs of the memory and the memory model is removed from the circuit. If the address lines are connected to the Data-out in the revised model, then the ATPG sets up the test by generating the appropriate data using the address inputs. During application of the test, the data from that memory location are written onto the Data-out lines. A defect

on the data lines will cause the wrong data to be loaded into memory during the preprocessing phase, whereas a defect on the address lines might escape detection.[20,21]

8.4.6 Implementing Scan Path

A scan path can be created by the logic designers who are designing the circuit, or it can be created by software during the synthesis process. If scan is included as part of a PCB design, the PCB designers can take advantage of scan that is present in the individual ICs used to (a) populate the PCB and (b) connect scan paths between the individual ICs. However, as will be seen in the following paragraphs, connecting ICs into a comprehensive scan solution can be a major challenge because, when scan is designed into the ICs, it is usually designed for optimal testing of the IC, with no thought given as to how it might be used in a higher-level assembly. Vertically integrated companies—that is, those that design both their own ICs as well as the PCBs that use the ICs—can design scan into their ICs in such a way that it is useable at several levels of integration.

For an IC designed at the register transfer level (RTL), scan path can be inserted while writing the RTL description of the circuit, or it can be inserted by a postprocessor after the RTL has been synthesized. A postprocessor alters the circuit model by substituting scan flip-flops for the regular flip-flops and connecting the scan pins into a serial scan path. Using a postprocessor to insert the scan path has the advantage that the process is transparent to the designers, so they can focus their attention on verifying the logic. However, when the scan is inserted into the circuit as a postprocess, it becomes necessary to re-verify functionality and timing of the circuit in order to (a) ensure that behavior has not been inadvertently altered and (b) ensure that delay introduced by the scan does not cause the clock period to exceed product specification.

When an ATPG generates stimuli for a circuit, it assigns logic values to signal names. However, it is not concerned with the order in which signal names are processed. That is because, when it is time to apply those values to an actual IC or PCB on a tester, a map file is created. Its purpose is to assign signal names to tester channels. The map file also accomplishes this for scan, the difference being that many stimulus values are shifted into scan paths rather than applied broadside to the I/O pins of the device-under-test (DUT). Whereas the stimuli at the I/O pins of an IC or PCB must be assigned to the correct tester channel, the scan stimuli must not only be assigned to the correct channel, but must also be assigned in the correct order.

This ordering of elements in the scan path is determined by the layout of transistors on the die. That order is identified during placement and route so that vectors generated by the ATPG can be applied in the correct order to the DUT. One job of the place and route software is to minimize total die area. So the order of scan elements is determined by their proximity to one another. Some constraints may be imposed by macrocells; for example, an n-wide scannable register may be obtained from a library in the form of a hard-core cell (i.e., a cell that exists in a library in the form of layout instructions), so its flip-flops will be grouped together in the same scan string.

If debugging becomes necessary when trying to bring up first silicon, some groupings, such as *n*-wide registers, may be easier to interpret when reading out scan cell contents if the bits are grouped. In addition to scan-cell ordering, the tester must know which physical I/O pins are used to implement the scan path: which pins serve as the scan-in, which serve as the scan-out, and which pins are used for test control.

Another tester-related task that must be considered during scan design is the application of vectors to the IC or PCB. The vectors are designed to be serially scanned into the DUT, and some testers have special facilities dedicated to handling serial scan and making efficient use of tester resources. One or more channels in the tester have much deeper memory behind the scan channels. While data on the parallel I/O pins are held fixed, scan data are clocked into the scan paths. Additional hardware may be available on the tester permitting control of the process of loading and unloading serial data in order to facilitate debugging of the DUT or of the test.

When testing scan-based designs with a tester that has no special provisions for scan path, it is necessary to perform a parallelize operation. When parallelizing a vector stream, each flip-flop in a scan path requires that a complete vector be clocked-in.

Example Assume that a device has nine input signals, four output signals, and ten scan-flops and that the input stimuli are 011001011. The output response is HLLH, the scan-in values are 1011111010 and the scan response is HHHHLHLLHL. Then the tester program for loading this vector might be as follows:

```
0 H 011001011 HLLH
1 H 011001011 HLLH
1 H 011001011 HLLH
0 H 011001011 HLLH
0 L 011001011 HLLH
0 H 011001011 HLLH
1 L 011001011 HLLH
0 L 011001011 HLLH
1 H 011001011 HLLH
1 L 011001011 HLLH
```

■■

In this tester program the stimuli applied to the I/O pins are repeated ten times. This represents a significant cost because there must be a large amount of memory behind every pin. This result is also somewhat less intuitive, in the event that it becomes necessary to debug test results, either when trying to get first silicon to work or when trying to improve yield.

One reason why parallelization is used is because companies often have large investments in expensive testers, and it is simply not practical to replace them. It becomes important to use them and amortize their cost over several products. One way to reduce the cost of test while using older testers is to implement multiple scan paths in the design. In the example above, if two scan chains were used and if each

of the scan chains were five bits in length, then the total number of vectors is reduced by half.

If there were a large number of scan vectors and if there were also a large number of scan bits, there may not be enough memory behind the tester channels to permit a complete test to be applied to the DUT. This argues for using multiple scan paths. Another argument for using multiple scan paths is the fact that the application of scan vectors is often done at a speed much slower than the intended operating speed of the DUT. When serially shifting in a large number of scan bits during test, a lot of switching takes place not only in the scan elements, but also in the combinational logic driven by these scan-flops. There is a potential for heat buildup, a potential that increases as the scan clock speed increases, introducing an unnecessary risk to the DUT.

Since added time on the tester represents added manufacturing cost for the DUT, it is desirable to apply the test as quickly as possible. With multiple scan paths, it is possible to reduce time on the tester. It has been pointed out that these considerations can also shorten the design cycle for designs being fabricated at a foundry.[19] The less critical the tester requirements for a design, the more flexibility the foundry has when scheduling the product on its test floor, since there may be more testers available that are capable of handling the assignment.

Multiple scan paths are usually implemented by sharing functional signals with scan signals at the I/O pins. At the output pins the test mode pin controls the multiplexing operation. The assignment of scan-flops to the multiple chains is often influenced by factors in addition to scan length reduction and the proximity of scan-flops to one another. Sometimes it becomes necessary to implement scan in designs that use multiple clocks, or where some flip-flops are clocked by positive clock edges and others are clocked by negative clock edges.

Consider a design with two clocks as shown in Figure 8.25. Assume for the sake of simplicity that all of the flip-flops are active on the positive edge. This circuit has three combinational blocks of logic, C_1, C_2 and C_3, and each of the two clock domains, CK1 and CK2, has two flip-flops. A feedback line exists from C_3 to C_1.

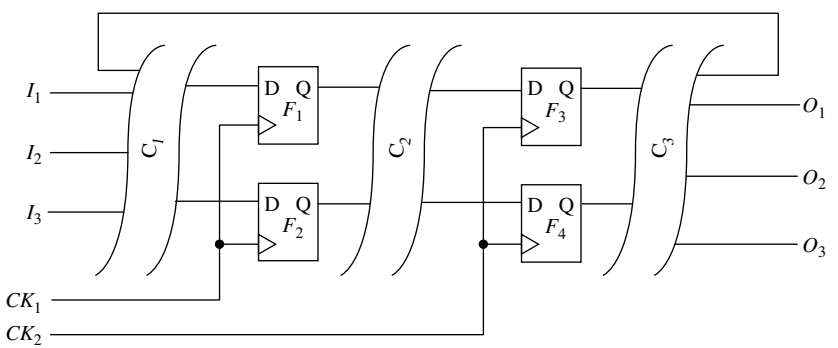


Figure 8.25 Circuit with two clocks.

The feedback line may be doing something as simple as updating a status bit in a register, or it may be doing something that has a pervasive effect on all or most of combinational block C_1. The important thing to note is that, because of the manner in which CK1 and CK2 are staggered, scan results become unpredictable. Consider the clocking scheme illustrated in Figure 8.26. Loading of the scan chains alternates, first scan chain 1 is clocked, then scan chain 2 is clocked. During this time the two chains are independent of one another, that is, the loading of one chain has no effect on the contents of the other.

When *scan_enable* goes low for a functional cycle, CK_1 is pulsed first, followed by CK_2. The ATPG specified data values in flip-flops F1 and F2 is based on the assumption that all of the flip-flops would be clocked simultaneously. But when CK_1 was functionally clocked, those values changed. Hence, the faults that were targeted by the ATPG may or may not actually be detected when CK_2 is pulsed. Many different complications can occur when multiple clock domains exist, depending on the feedback lines. For that reason it is recommended that fault simulation be performed to verify the fault coverage when there are multiple clock domains.

Another problem that often has to be dealt with is the presence of both positive and negative edge clocking. If both positive and negative edge triggered flip-flops are to be placed in the same scan chain, it is recommended that the negative edge triggered flip-flops be placed at the beginning of the scan chain. Another possible solution, assuming that the clock period is of sufficient duration, is to complement the clock. However, in large circuits there is seldom, if ever, excess time in a clock period.

The lockup latch is another solution to the problem of mixed clocks. In fact, the lockup latch can help to alleviate many problems, including clock skew. Skew is an observed difference in time between two events that are supposed to occur simultaneously. When a clock is driving many hundreds or thousands of flip-flops, those flip-flops may possess minute variations in their behavior. A possible effect is a difference in timing between the flip-flops in a scan chain. Because two flip-flops that are logically adjacent may be physically distant from one another, the skew may be sufficiently pronounced as to cause the wrong value to be loaded into a flip-flop.

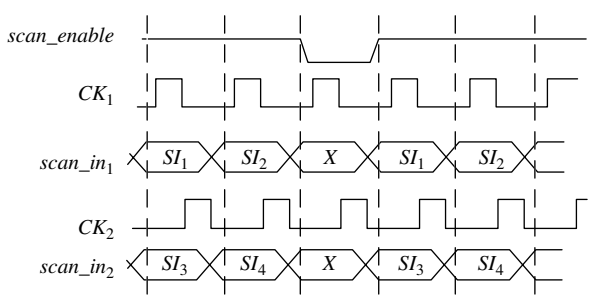


Figure 8.26 Clocking sequence.

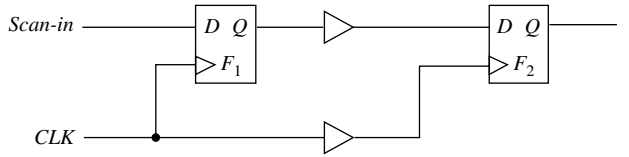


Figure 8.27 Clock skew.

Consider the circuit in Figure 8.27. There is a delay element inserted in the scan connection between the Q output of F_1 and the D input of F_2. There is another delay in the wire driving the *CLK* input to F_2. These delays represent resistance in the wire runs, as well as capacitance between the wire runs and other circuit elements. Denote by T_p the total elapsed time from when F_1 recognizes an active clock edge to when the signal at the D input of F_1 propagates through F_1 and through the wire connecting F_1 to F_2. Then T_p must exceed $T_h + T_{skew}$, where T_h is the hold time of F_2 and T_{skew} is represented by the delay in the clock line. If the clock skew is excessive, the new value loaded into F_1 makes its way to the D input of F_2 before the clock edge appears and causes the new data in F_1 to be loaded into F_2.

Now consider the circuit depicted in Figure 8.28. A lockup latch L_2 is interposed between F_1 and F_3. When *CLK* is low L_2 is enabled, or transparent. When *CLK* goes high the enable *EN* of L_2 goes low, so the data at the output of F_1 is held for an extra half period. This effectively adds a half clock of hold time to the output of F_1. This solution can be used to solve clock skew, as well as to connect scan elements that are in different clock domains. It is also recommended for scan chains that contain both positive and negative edge clocks.

Even when a solution exists, such as the lockup latch, it is still advisable to group flip-flops according to their clocking domain and edge. For example, a lockup latch makes it possible to connect both positive and negative edge-triggered flip-flops in the same scan chain, but, unless there is excessive clock skew, the chain should only need a single lockup latch if all the negative edge flip-flops appear at the beginning of the chain and all of the positive edge flip-flops appear after the negative edge flip-flops. And, of course, when multiple scan chains are used, it is advisable to make all

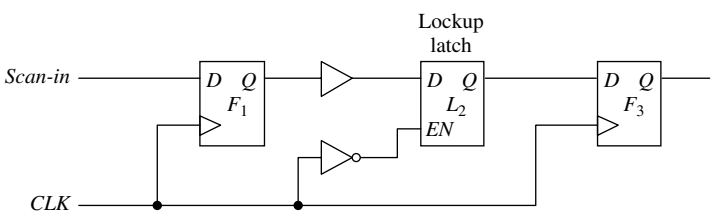


Figure 8.28 The lockup latch.

of the scan chains of equal or near equal length. When different size chains occur in a design, then the stimuli must be lined up such that all of the chains are loaded correctly.

Because testers tend to be quite expensive, it is desirable to apply test programs in the shortest possible time, in order to maximize throughput on the tester. One way to accomplish this is to reduce, as much as possible, the number of vectors applied to the circuit. However, vectors cannot simply be discarded without impairing the quality of the test. In Section 7.9.6, static and dynamic test pattern compaction were discussed at length. Compaction is especially attractive for scan test programs where pairs of vectors have to be considered, in contrast to sequential test programs where two or more sequences of n vectors, for arbitrary n, have to be merged without conflict.

Another strategy for reducing test vector count in scan circuits is *test set reordering*. In this scheme the set of vectors is fault-simulated and then reordered so that those yielding highest-fault coverage occur first and those with the smallest number of detections occur at the end. Then the reordered set of vectors is fault simulated. Often the small number of faults detected by the vectors occurring at the end are detected by other vectors occurring earlier in the sequence. Those vectors that don't add to the fault detection can be discarded. This procedure may produce useful results in two or more iterations, and the resulting savings in test time may be especially useful for high-volume commodity ICs. If the total number of vectors exceeds the number that the tester can handle, this scheme can help to determine which vectors to keep and which to omit from the test program.

Another potential savings in test time may flow from the use of scan chains of unequal length. Conventional wisdom would argue for an assignment of flip-flops so that all scan chains are of equal or near-equal length. However, it has been demonstrated that scan chains of unequal length can sometimes be more effective, resulting in up to a 40% reduction in test time.[22] This is based on the observation that some flip-flops are much more active than others, both functionally and when testing a circuit. It may be the case that a block of logic—for example, an ALU or some other deep data path circuit—requires a large number of vectors, but the number of scan-flops used to test the block is quite small. On the other hand, there may be a large number of scan-flops involved in control logic. The control logic may be quite shallow, perhaps containing only two or three levels of logic from input to output scan-flops.

One way to determine assignment of scan-flops to scan chains is by ordering the scan-flops according to the number of times that each scan-flop is assigned a known (0 or 1) value. If a small number of scan-flops are assigned almost always, whereas the remainder are assigned values infrequently, then the scan chains can be partitioned based on the frequency of the assignments.

Example Assume that a circuit contains 500 scan-flops, a total of 600 scan vectors are created by the ATPG, and that a maximum of two scan chains are permitted for the design. Assume also that a subset of 50 scan-flops are assigned values for at most 500 of the 600 scan vectors and that the remaining 450 scan-flops are assigned

values for at most 200 of the 600 scan vectors. If the scan-flops are divided arbitrarily into two chains of 250 scan-flops each, and 600 vectors are applied to each, then $600 \times 251 = 150{,}600$ scan plus functional clocks are required to fully test the circuit.

Now consider the situation where the scan chains are partitioned so that one scan chain contains 50 scan-flops, and the other contains 450 scan-flops. The larger chain requires $450 \times 201 = 90{,}450$ clocks. The smaller scan chain requires $50 \times 301 = 15{,}050$ clocks (200 vectors are scanned in concurrently with the larger chain). The total number of clocks is 105,500, a significant reduction from the case where both chains are of equal size. ■■

8.5 THE PARTIAL SCAN PATH

The use of full-scan provides total controllability and observability. Unfortunately, it is not always feasible to employ a full-scan test methodology. Some designs are constrained by area and/or performance requirements, and some circuitry is not testable by scan. Memory blocks, including cache memory, scratchpad memory, fifos, and register banks, which in earlier days were contained in stand-alone chips, now share a common die with logic. These memories are normally excluded from the scan chain and tested using memory BIST, as pointed out in Section 8.4.5. Analog circuitry represents another problem for scan. Memory and analog circuits must be isolated from the digital logic, circuit partitioning becomes critical, and testing strategies for memories and random logic must now coexist.

Sometimes full-scan is not an option because there is not enough room on the die and the inclusion of additional logic necessitates migrating to a larger die size. This could be the case in instances, such as gate arrays, where the die are available in discrete increments. Multiple clock domains present another problem to full scan, as was seen in the previous section. If a very small percentage of the storage elements exist in a separate clock domain, it might be practical to completely omit them from scan.

When full-scan is not an option, partial scan can be used to test the circuit. In this mode some, but not all, of the flip-flops are stitched into a scan path. The partial scan chain can include flip-flops from just a few of the more troublesome circuits, such as status registers, counters, and state machines, to use of scan for everything except a few timing-critical signal paths. Testability analysis tools such as SCOAP can help to determine where partial scan would be most effective. Another way to select scan-flops is to let the ATPG select those flip-flops that it is not able to control or observe. Additional methods, discussed in the following paragraphs, select scan-flops based on other criteria in order to improve fault coverage or to reduce die area dedicated to scan or test time.

A drawback to partial-scan, depending on how it is implemented, is that it negates one of the major benefits of scan. If a complete scan-path exists, ATPG is tremendously simplified, there is no need for an ATPG with sequential test pattern generation capability. A partial scan path that excludes some sequential elements but

leaves others in the circuit may require an ATPG with sequential circuit processing capability.

The benefits of partial scan depend to some extent on how well the ATPG is implemented. If the ATPG can handle latches, combinational loops, and feed-forward or loop-free sequential logic (cf. Section 5.4), it has been shown that it is possible to achieve acceptable fault coverage in the neighborhood of 95% on large circuits with about half of the flip-flops included in scan chains.[23]

When partial scan is being considered, the important question that must be answered is, Which flip-flops should be scanned? The answer to that question, in turn, will depend on the answers to the following questions:

How much increase in die size can be tolerated?

Can performance degradation be tolerated?

What is the fault coverage objective?

What are the capabilities of the ATPG?

How many test vectors can the tester handle?

The attraction of full scan lies in the fact that high-fault coverage for structural defects is relatively easy to obtain, test programs can be generated in a predictable amount of time, and there is some control over the size of the test program. Objections to scan have always been based on the fact that it adversely affects die size and performance. Partial scan makes it possible to mitigate some of these concerns, such as the adverse impact on die size, and by proper selection of flip-flops to be included in the scan chain it is often possible to avoid, or at least minimize, performance degradation. This stems from the fact that critical flip-flops—that is, those with critical timing—can be identified and excluded from the scan path. This consideration helps to partially answer the question raised above, at least in the sense of identifying flip-flops that should not be scanned. A number of strategies have been devised over the years to help complete the selection process.

When the decision is made to employ partial scan, it must be decided whether it is actually going to be partial scan—that is, one in which just a few flip-flops are scanned—or whether it is going to be *almost-full scan*. Sometimes an ATPG fails to create an effective test for a sequential circuit due to the presence of a small amount of circuitry that is difficult to control, such as large counters or complex state machines. In these cases, it may be possible to put the troublesome flip-flops on a separate clock, or on a separate branch of a clock tree, so they can be loaded while the remainder of the circuitry is held fixed in its current state. In Figure 8.29 the values in the flip-flops on the right side of the circuit are held fixed if test control *TC* is set to 0, while the partial scan flip-flops on the left side are loaded by means of the *scan-in* input. In normal functional mode $TC = 1$, so all flip-flops are clocked by *CLK* and the scan-flops receive their data from the combinational logic by means of the multiplexers at their inputs.

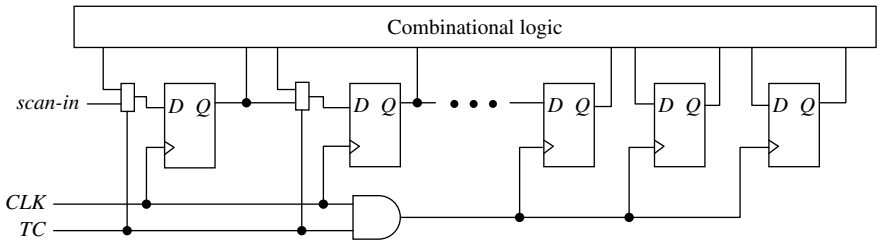


Figure 8.29 Partial scan clocking.

The ATPG treats the scan-flops as primary inputs and primary outputs, just as in full scan. However, the goal is to try to avoid using them too often. The scan-flops may be members of a state machine that is difficult to control, but, once loaded, other sequential circuitry may be only mildly sequential, permitting the ATPG to achieve acceptable fault coverage. It may be the case that the state machine is not difficult to control, but perhaps some status signals that control its transitions are themselves too difficult to control, in which case the partial scan can be used to select values for the status signals.

The almost-full-scan approach, in contrast to the partial scan, is often implemented by starting with full scan, and then removing flip-flops based on performance or area criteria. For example, there may be a small number of flip-flops that are in critical timing paths, such that it is impossible for a device to meet its performance goals if they are scanned. These performance goals may be mandatory, as in the case of a device that absolutely must perform correctly at a designated frequency in order to satisfy an industry standard, without which it would have no value in the marketplace. The solution is to identify and remove from the scan chain those flip-flops that are in the critical paths. In this mode a high percentage, often 80–90% or more of the flip-flops, are scanned.

During test generation the flip-flops that are not in the scan path are clocked exactly like the flip-flops that are serially connected into scan chains. However, their D-inputs are driven not by scan-flops but, rather, by functional logic. As a result, these inputs are being constantly stimulated by random functional data that originates at the scan-flops and passes through combinational logic. This is sometimes referred to as "destructive partial scan" because in the process of scanning new data into the scan chain, data in those flip-flops that are not part of the scan chain is destroyed.

The wildly fluctuating input to these flip-flops causes their values to be unpredictable, so they are treated as X-generators; that is, they generate an X state. In other respects the implementation may resemble full scan. Fault coverage is reduced to the extent that logic driving only these flip-flops is unobservable, as depicted in Figure 8.30. In addition, flip-flops that generate Xs cause other faults to be, at best, only potentially detectable. For example, the top input to gate *D* requires a 0 to test

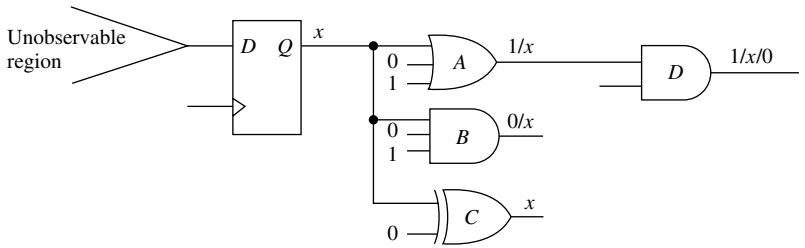


Figure 8.30 Undetectable faults.

for a SA1, but it is not possible to apply a 1 to that input. Note that this analysis can quickly identify the pervasive effects of state machines and other control logic that drive a great deal of other logic.

Using simple network analysis tools it is possible to measure, for each flip-flop, the number of faults that lie in the unobservable region, and it is possible to count the number of faults that can only be possible detects. These numbers can be generated for each flip-flop in the circuit and used as a basis for deciding which flip-flops will be excluded from the scan chain. If, for example, 10% of the flip-flops are to be excluded from scan, then the undetectable faults in their unobservable regions, and those in the fanout from these flip-flops, can be summed to give an approximate count of the total number of undetectable faults in the circuit (note that unobservable regions may overlap). This gives an approximate upper limit on achievable fault coverage. This upper limit can be used to decide whether the approach is acceptable, or whether some other solution must be pursued.

If an upper limit on fault coverage reveals that the method cannot achieve an acceptable fault coverage goal, then one possible alternative is to employ an ATPG with some sequential capability. In this mode the ATPG can exercise the functional clock an arbitrary number of times between scan shifts, with the result that some nonscannable flip-flops may eventually assume known values and it becomes possible for otherwise undetectable faults to become detected. This differs from the partial scan scenario just described in that the unscanned flip-flops start a sequence with unknown values, but can be driven to a known value during a sequence.

Yet another alternative is to employ design verification vectors to the extent that they are useful. These may cause 60–70% of the faults to be detected with a small functional test. The functional test program can be truncated when it reaches diminishing returns. At that point the method just outlined can be employed, but the flip-flops can now be ranked according to how they affect observability and controllability of the undetected faults. The result may be quite different from the result obtained using the complete fault list, and it may be possible to remove a significantly greater number of flip-flops from the scan chain while achieving acceptable

fault coverage. This approach has an additional advantage, as pointed out in Section 7.2, of detecting faults during a dynamic functional test that a static, fault-oriented scan test may miss.

A scan approach called Scan/Set was described in 1977.[24] This method provided parallel/serial flip-flops that could be loaded and read out via a scan path, but the registers were separate from the functional logic. They therefore had somewhat less impact on the performance of the functional logic. The Set feature, which loaded operational flip-flops from the Scan/Set flip-flops, was used only for flip-flops judged to be difficult to control. Multiplexers routed signals to the output pins, and several internal points could be selected for observation by the multiplexers. Ad hoc design rules existed as part of the system. These rules both prohibited certain design practices and helped to select nodes to be scanned or set.

An early paper describing partial scan removed scan-flops from the circuit model, then analyzed the remaining circuit for complexity.[25] One of the rules for the system prohibited the remaining, non-scan circuit from having a sequential depth exceeding three, meaning that it must be possible to drive any flip-flop to a given value in no more than three time frames. A single clock controlled both the scan and non-scan flip-flops. Fault simulation of the complete circuit, including every scan clock, was performed. This had the advantage that it was possible to predict the values in all of the flip-flops, regardless of whether or not they were in the scan chain. However, even for the relatively small circuits of that era, this led to long simulation times.

The frequency approach was another method for choosing scan-flops.[26] Design verification vectors were first used to exercise the circuit functionally and eliminate from further consideration the faults that were detected by these vectors. During this phase of the operation, the functional test would be truncated at a point of diminishing returns—that is, at that point where many functional vectors were required to set up the circuit in order to detect very few additional faults.

PODEM was used during the frequency approach to target undetected faults. It generated all possible tests for targeted faults. From these tests the one requiring the smallest number of scan-flop assignments was chosen. A record was kept of the flip-flops required by each test. Then the goal was to select, for a given number of flip-flops, a set of tests that covered the largest number of faults. If coverage was insufficient, additional flip-flops could be added to the partial scan chain. This would allow additional tests to be included, thus improving fault coverage. An alternative approach could also be considered. If a scan chain requires too much die area, or causes the test length to exceed some threshold, this approach could be used to eliminate the least productive flip-flop(s) from the scan chain.

In Section 8.4.6 it was noted that, for full-scan implementations, scan-flops could be grouped into those of high usage and those of low usage. By grouping scan-flops and constructing scan chains accordingly, it was possible to achieve a significant reduction in the number of clocks required to apply a test. A somewhat similar approach was used to group flip-flops for a partial scan solution.[27] This approach assumes the existence of a partial scan chain and the use of an ATPG to create sequences, or blocks, of vectors to test a target fault. Two observations are made regarding these blocks:

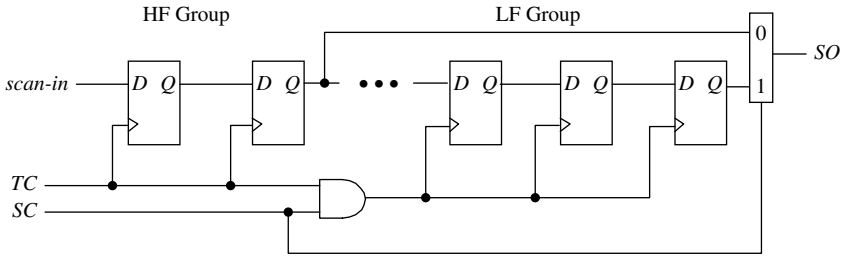


Figure 8.31 Scan control for vector reduction.

1. There is a broad distribution in the frequency of usage of scan locations in a partial scan circuit.
2. The vast majority of fault detections occur on the last vector of each block.

The scan-flops are divided into two groups, the high-frequency (HF) set, and the low-frequency (LF) set. Whether a scan-flop falls into the HF or LF set depends on its frequency of usage during test pattern generation. Scanning out the HF or both HF and LF is accomplished by means of the circuit in Figure 8.31. When *SC* is set to 1, both the LF and the HF groups are selected by the multiplexer. When *SC* is set to 0, only the HF group is passed to the scanout pin *SO*.

During test pattern generation a fault is selected as the target, and a block of vectors is generated to test this fault. On the first vector of this block, the entire partial scan chain is scanned out in order to detect the targeted fault from the previous block. For the remaining vectors in the block, if a scan-flop in the LF group changes, set *SC* to 1. If a scan-flop in the HF group changes, but no scan-flop in the LF group changes, set *SC* to 0. If no scan-flop in either group changes, do not scan, just apply the primary inputs. It has been reported that this approach has resulted in reductions of 60–70% in the length of test programs. This reduction in test cost must, of course, be weighed against the added cost due to an increase in die size.

In Section 5.4 we discussed the complexity of test pattern generation. It was pointed out that a cycle-free sequential circuit—that is, one in which there are no feedback paths—was not much more difficult to test than a combinational circuit. Occasionally, while backtracking, the ATPG would have to remember that some flip-flops required different logic values in different time frames. This observation about acyclic, or feed-forward, sequential circuits suggests that perhaps, for partial scan, the best flip-flops to select for scan are those that can break up cycles and reduce the circuit to a feed-forward sequential circuit.

Consider the S-graph in Figure 8.32, where the nodes represent flip-flops and the arc represents connections between flip-flops. The vertices F_1 through F_4 represent flip-flops, and the arcs represent combinational logic connecting the flip-flops. This could conceivably represent a one-hot encoded state machine with four flip-flops. If any one of the flip-flops F_1 through F_4 is scanned, then for test purposes this subcircuit is acyclic. As mentioned above, the requirements on the ATPG that processes

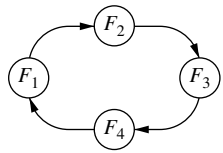


Figure 8.32 S-graph of circuit with four flip-flops.

acyclic circuits are greatly simplified. It is estimated that, in general, about 25% of the flip-flops must be scanned in order to reduce a circuit to acyclic form.[28]

Without additional knowledge about the circuit, the choice of which of the flip-flops F_1 through F_4 should be chosen for inclusion in the partial scan path is arbitrary. However, it often happens that some choices may be excluded because the flip-flop lies in a critical path, and a scan-flop causes propagation time to exceed the clock period. Another factor that may be considered is the effect the scan-flop has on circuit layout.[29] Some routing channels may be too congested to accommodate the scan overhead.

Test length is yet another variable that can be taken into account when choosing flip-flops for partial scan. It is possible to create a circuit that is feed-forward, or acyclic, but the sequential depth is excessive.[30] As a result, after loading the partial-scan chains, a large number of functional clocks may be needed to propagate the test sequence forward to an output. Careful analysis may reveal that converting just a few additional flip-flops to scan-flops will significantly reduce the test length, so that overall product cost (i.e., cost of die plus cost of test), is reduced. It has been suggested that an upper limit on the number of scan-flops be established. Then, if the number of scan-flops required to break all cycles is less than the number permitted, SCOAP or a similar such testability analysis tool can be used to select additional flip-flops for inclusion in the scan chain.[31]

It may be possible to reduce test sequence length by establishing design rules.[32] While this may not be an acceptable general solution, there may be instances where choices exist for implementing macrocells in a library, and sequential test depth may be one of the parameters used to determine which choice is adopted.

8.6 SCAN SOLUTIONS FOR PCBs

The in-circuit tester (cf. Chapter 6), was an effective means for identifying problems on printed circuit boards when dual in-line packages (DIPs) were the prevailing packaging technology. However, the industry began gradually to move away from DIPs during the 1980s, and newer packaging technologies have made it much more difficult to access I/O pins with the in-circuit tester. Recognizing this, electronics companies began looking for alternative methods to detect faults on PCBs. The following defect distribution for PCBs was compiled by Hewlett-Packard:[33]

37% — Opens
22% — Missing or wrong chip
19% — Faulty analog device
14% — Dead ICs
7% — Shorts
1% — Fixture

In this list, opens are the most frequently occurring type of defect. Other studies come up with different numbers, but the profile generally follows the same trend. Opens can be troublesome on PCBs employing ball grid array (BGA) technology. A solder re-flow technology is used in which solder balls are placed on the bottom of the IC. The IC is positioned on the PCB and reheated. The solder balls then melt and make contact with metal pads on the PCB. Failure to make contact with the PCB can result in opens. Opens can also occur if wave soldering is employed after the BGA chip(s) are attached to the PCB. There is a tendency to suspect opens in the BGA when the PCB fails to work properly. However, it has been reported that 75% of all suspected solder joint failures associated with BGAs have turned out not to be the problem.[34] Removing BGAs that are fault-free results in many PCBs being needlessly damaged.

The NAND tree is an effective DFT methodology for detecting opens caused by bad solder joints at IC pins. However, it is not effective at detecting the other problems in the above list. A more general solution for detecting a wider array of defects was initially proposed by a European group, known as the JTAG (Joint Test Action Group). They were eventually joined by companies in the United States. Working with the IEEE, they developed the IEEE 1149.1 boundary scan standard. In this section we first look, briefly, at the NAND tree and then look in detail at boundary scan.

8.6.1 The NAND Tree

The NAND tree, shown in Figure 8.33, is used to provide a test for continuity between I/O pins and the pads on a die. A NAND gate is placed between each I/O pin and its corresponding pad. Output pins are modified through the use of a tri-state driver so that they can be isolated from the pad during the test. The signal called NTST_, which controls the NAND tree test, is inactive when high. When it is set low, it isolates the output pad from the pin and also disables the output mode of the bidirectional pin. All of the pins that are included in the NAND tree are initially set low. The NAND tree output then expects the initial output response to be low (logic 0).

The input assigned the number (1) is set high on the next clock cycle. The NAND gate that it is driving goes low and, as a result, the NAND that it is driving goes high. On the next clock cycle the input to the cell labeled (2) is set high. Its corresponding NAND goes low, causing the NAND in cell (1) to go high, and that causes the NAND

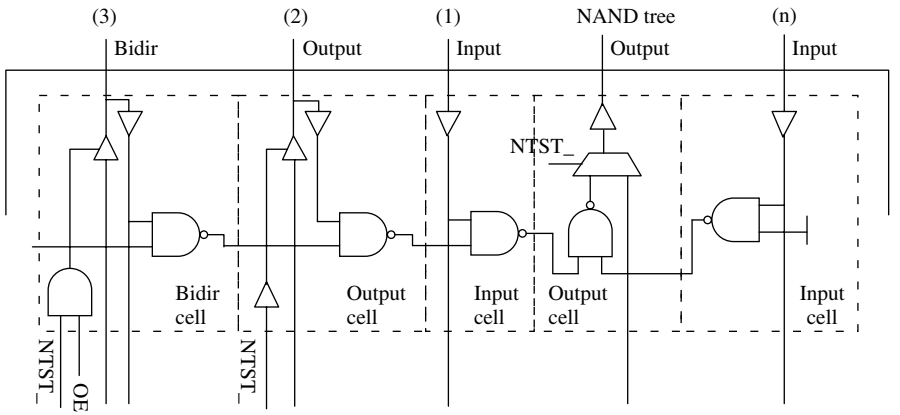


Figure 8.33 The NAND tree.

tree output to go low. This continues until all of the cell inputs have been set high, causing the output to alternate between 1 and 0. If there is an open between any of the input pins and its corresponding pad on the die, the output waveform goes flat, either a constant 1 or a constant 0.The number of pulses that appear at the NAND tree output can reveal which input is defective.

8.6.2 The 1149.1 Boundary Scan

The IEEE 1149.1 standard[35] goes beyond the NAND tree. Like the NAND tree, it can detect opens at the I/O pins, but it can also identify shorts between I/O pins, as well as opens and shorts on the PCB. It can identify bad ICs or the wrong IC in a particular socket on the PCB. The boundary scan registers can be connected to an internal scan path or BIST circuit, while isolating the IC from the board, making it possible to test the internal circuits of the IC while it is mounted on the PCB. A complete IC test may not be practical via the 1149.1 standard, but a few patterns from a scan test can usually get high coverage (cf. Section 7.7.1). By being able to identify defective ICs on the PCB, an internal test can often make it economically feasible to salvage PCBs that fail board test.

It must be pointed out that 1149.1 can be applied hierarchically, to any level of integration. The discussion that follows is based on ICs mounted on PCBs, but could have been centered, without loss of generality, on a complex system made up of multiple PCBs. The value of this observation stems from the fact that, with boundary scan, it is possible to standardize test throughout an entire hierarchy, from chip to board to system test. It should also be pointed out that boundary scan, while a next-generation replacement for in-circuit testers, does have its own shortcomings. Test data are serialized, causing longer test times. Because of this, there are potential problems with keeping dynamic logic alive, as well as potential

problems with overdrive limits. Also, power must be applied when testing devices. The in-circuit tester can detect many defects, such as shorts, without applying power, thus reducing the likelihood of damaging the PCB. On the other hand, an in-circuit tester can destroy the very device it is attempting to test when it overdrives the IC.

The 1149.1 standard consists of a test access port (TAP), a set of registers, and a state machine. The TAP is a set of dedicated I/O pins used to access test mechanisms on the IC or PCB. The set of registers includes the following: a boundary scan register that implements a scan path around the periphery of the chip, an identification register that contains a unique code identifying the chip, an instruction register, and a bypass register. The state machine controls the operation of the various registers. It selects registers and causes them to be shifted or updated.

Figure 8.34 shows four ICs mounted on a PCB. Various interconnections run between the I/O pads of the ICs. The bold lines identify the boundary scan register (BSR). The BSR begins at the PCB input labeled TDI (test data input). It winds its way through the I/O pads of each IC, eventually reaching the PCB output labeled TDO (test data output). While the figure shows all of the ICs connected into the boundary scan ring, it is not unusual to have a PCB in which some, but not all, of the ICs are boundary scannable. The 1149.1 standard takes that into account and was designed to accommodate such situations.

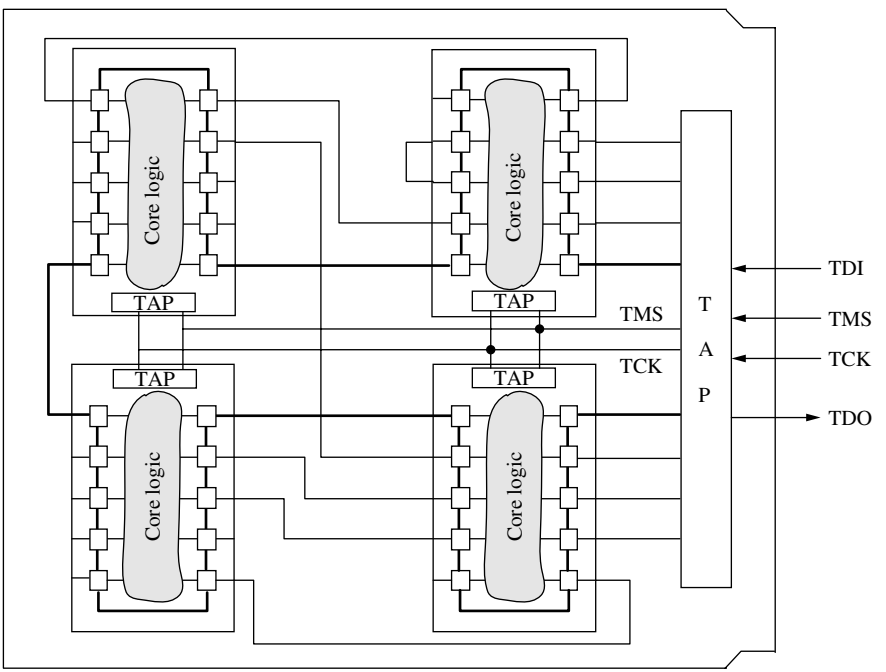


Figure 8.34 PCB with IEEE1149.1 boundary scan.

Two additional signals are shown in Figure 8.34, the test clock (TCK) and the test mode select (TMS). These two signals, which are distributed to a TAP controller on each individual IC, control the state machine found in each of the TAP controllers. The state machines, in turn, generate signals that control the boundary scan register, as well as the identification register and other registers. Before going into detail about the action of the state machine, we first look at a typical boundary scan cell.

The boundary scan cell shown in Figure 8.35 is a typical implementation suggested, but not mandated, by the IEEE1149.1 standard. This cell can be used at either an input pin or an output pin. If it is connected to an input, then Din represents a signal from outside the chip, and Dout represents the signal driving the internal circuits of the chip. The Mode input controls the routing of data through the cell; if Mode is 0, then data external to the chip pass straight through the multiplexer. This is the normal, functional mode. When Mode is 1, the boundary scan register is performing a test-related function, which may involve shifting or capturing data. Different mode-control signals may be used for input and output pins of the component, and the signals are derived from the instruction in the instruction register.

The ShiftDR, ClockDR, and UpdateDR signals are generated by the state machine and control the behavior of the cell. There are counterparts to these signals with the names ShiftIR, ClockIR, and UpdateIR. They are used when the cell is part of the instruction register. The ShiftIn signal may be connected to the TDI signal or to the ShiftOut signal of a neighboring boundary scan cell. The cell contains two registers, CAP and UPD. CAP is used to capture signals from Din or from a previous boundary scan cell, depending on the value of ShiftDR. The ClockDR signal from the TAP controller clocks the value into CAP. After all the CAP registers have been updated, either in parallel from Din or serially from ShiftIn, an UpdateDR signal clocks the contents of the CAP registers into the UPD register where, if the Mode signal is set to logic 1, the values can all be presented simultaneously to the Dout signals. The 1149.1 standard includes other suggested implementations of the cell. For example, if an I/O pad is to be used as an input only, then the UPD register and the mux driving Dout can be eliminated. Such a cell will support signal capture only.

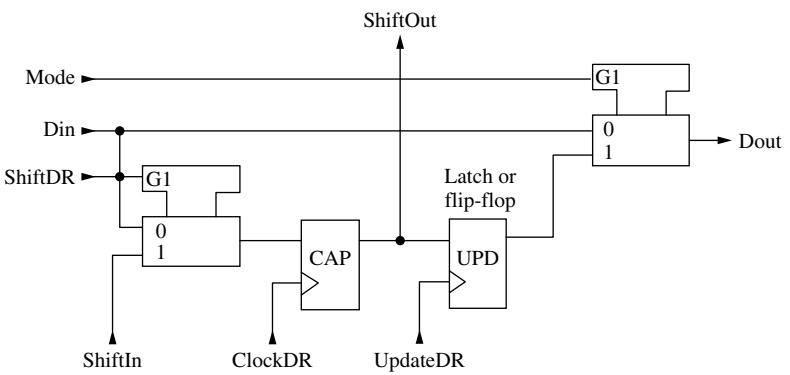


Figure 8.35 Boundary scan cell.

Figure 8.36 is a block diagram showing the relationship between the various functional parts that go into the making of a boundary scan device. The previously mentioned input signals are accompanied here by another signal, TRST*, a test reset signal (the asterisk denotes active low). The TRST* signal is optional. When present, it serves as an active low reset for the TAP controller. It must not be used to reset any of the system logic in the circuit. There are four test data registers shown in the diagram, but the *design-specific test data registers* could, in practice, represent any number of registers. The boundary-scan register and the bypass register are mandatory, and they are shown in solid lines. The device identification register and the design-specific test data registers are optional, and they are enclosed in broken-line boxes.

The signal at the TDI pin can go to the instruction register or to any of the four test data registers. The TAP controller determines whether the instruction register or a test data register receives the data. The first step in using IEEE1149.1 is to load the instruction register. After it has been loaded, the instruction register controls the mode signals in the boundary register cells, and in that way it determines which of the test data registers receives data from the TDI input.

The identification register is used to verify that a PCB has been populated with the correct IC. It can also be used to verify that the correct version of a chip has been used; or, in those cases where a part is manufactured by two or more vendors, the identification register can identify the vendor. It may be the case that several versions of a PROM exist. By scanning out the identification register, it can be determined if the PROM with the correct personality has been used on the PCB. The identification register, when implemented, is 32 bits in length. The high-order 4 bits, 31 to 28, contain the version number. Bits 27 down to 12 contain the part number. The next 11 bits contain the manufacturer identity, and bit 0 is always a logic 1.

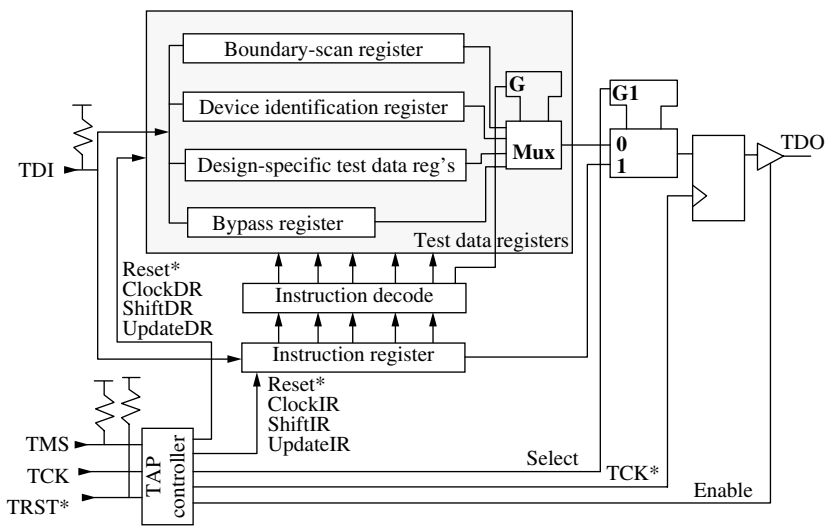


Figure 8.36 Block diagram of a boundary-scan device.

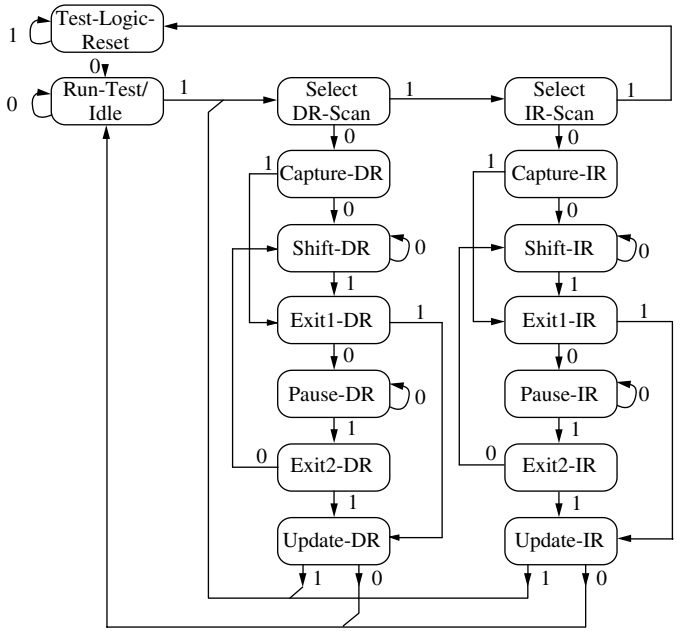


Figure 8.37 State diagram of the TAP controller.

The design-specific test data registers may represent internal scan paths or other DFT constructs, such as shadow registers, and so on. In this way, IEEE1149.1 facilitates testing of a device while it is mounted on a PCB. The purpose of the bypass register is to make it possible to access a particular IC on the board while minimizing the number of clock pulses required to pass through other ICs. Consider again Figure 8.35. When the user targets a particular IC for testing, all of the other ICs can be put into bypass mode. Since the bypass register is a single flip-flop, only one clock pulse is needed to pass data through it. Thus the target IC can be accessed with significantly fewer clock cycles.

The state machine transitions are illustrated in Figure 8.37. At power-up, or at the presence of a logic low signal on TRST*, the state machine enters the Test-Logic-Reset state. The state machine remains in this state as long as TMS is at logic 1. The Instruction Register is also reset at power-up or at the occurrence of a logic low signal on TRST*. As a result, the 1149.1 circuitry is made to appear transparent and the circuit in put into its normal, functional state. Note also that the asterisk (*) is used in the 1149.1 standard to denote an active low signal. So, the TCK* emanating from the TAP Controller in Figure 8.37 clocks the flip-flop driving Dout on a falling edge of TCK.

In order to employ boundary scan, the TAP controller must leave the Test-Logic-Reset state. This requires a positive edge on TCK while TMS is set to logic 0. Then, the state machine enters the Run-Test/Idle state. The TAP Controller remains in this

state as long as TMS is low. The circuit may simply remain idle, or the functional logic may exercise a built-in-self-test. When TMS goes to logic 1, and a positive edge is applied to TCK, the state machine transitions to the Select DR-Scan state. This is a temporary state from which the state machine transitions to either the Capture-DR state or the Select IR-Scan state, depending on whether the individual programming the TAP controller wants to load a data register or an instruction register. The Select IR-Scan is another temporary state from which the state machine either returns to the Test-Logic-Reset state, or it goes down the alternate route through the state machine.

Note that the two paths through the TAP controller are identical, with the exception that actions in the left leg of the transition graph apply to the selected test data register, while in the right leg the actions apply to the Instruction Register. As the TAP controller transitions through the various states, the ClockDR, ShiftDR, UpdateDR, ClockIR, ShiftIR, and UpdateIR signals are generated at appropriate times in order to implement various instructions.

Notice also that in each leg of the state transition graph, the second state is either Capture-DR or Capture-IR. From the capture state there is a transition to either a Shift state or an Exit state. The Capture state is used to load, or capture, parallel data, whereas the Shift state is used to serially shift data into the register labeled CAP. After the registers have been loaded, either through a parallel capture or a serial shift process, their contents can then be clocked into the update register (labeled UPD). This can be seen in Figure 8.36 where the ShiftDR and ClockDR signals permit data to be serially shifted from ShiftIn to ShiftOut. Alternatively, the ShiftDR and ClockDR can be conditioned to capture the value present at Din and clock it into the CAP register.

The signals just mentioned, together with the Instruction Register, are used to implement seven instructions. Three of them are mandatory; that is, they must be supported in order to be in compliance with IEEE1149.1. The three mandatory instructions are Extest, Bypass, and Sample/Preload. The optional instructions include Intest, Runbist, Idcode, and Usercode.

The Extest instruction is used when testing circuitry external to the IEEE1149.1 compliant device. Extest has three functions:

Stand-alone	Tests connection from BSR to the circuit board.
Interconnect	Tests connections from one boundary scan device to another.
Cluster	Tests circuitry (non-scan devices or clusters) mounted between one boundary scan device and another.

The Bypass instruction makes use of a single shift register, called the Bypass Register, which is placed between the TDI and TDO pins. Its purpose is to provide a minimum-length serial path through an IEEE1149.1 compliant device to another selected IEEE1149.1 compliant device during test or debug operations.

The Sample/Preload instruction provides two functions:

Sample	Sample data during normal circuit operation.
Preload	Load an initial data pattern at the latched parallel outputs of the BSR cells.

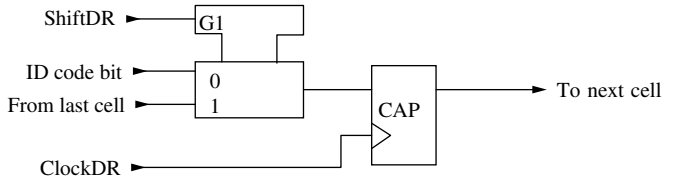


Figure 8.38 Device identification register cell design.

The Intest instruction supports static (slow speed) testing of the internal logic of a device. Test data are loaded onto the latched parallel outputs of the BSR cells using the Preload function. During this test, the device is isolated from the PCB input pins.

Runbist causes a device to run a self-test. The TAP controller is in the RUN-TEST/IDLE state during this test. At the conclusion of self-test the results are shifted out through TDO. During self-test the device is effectively isolated from the board because the device input and output pins are inactive.

The Idcode instruction provides access to the identification register in order to determine the identity of a component. A suggested implementation of the Identification register, from the 1149.1 standard, is shown in Figure 8.38. The ShiftDR signal is first set to 0 to load a hardwired ID code bit into the CAP flip-flop. Then, ShiftDR switches to 1 to facilitate shifting out of the ID on successive pulses of ClockDR. The Usercode allows a user-programmable identification code to be loaded into or shifted out of a device for examination. It is essentially an extended function of Idcode, for programmable devices. For this function to be valid, an identification register must be implemented for the IC.

Operation of the Preload instruction is illustrated in Figure 8.39. During this instruction the boundary scan registers are loaded without interfering with the existing state of the chip. This is particularly useful if the chip in Figure 8.39 drives two or more chips, and it is necessary to completely establish the state of the I/O pins before enabling these values onto the outputs. For example, suppose several memory chips drive a bus, but only one of them is permitted to be active at any time. The Preload permits all of the CAP flip-flops to be loaded, and then the UPD flip-flops are simultaneously loaded with the values in the CAP flip-flops. In this way, one of the destination memory chips is selected, and the others are deselected.

The bold lines indicate the path along which data flow during operation of the Preload instruction. The ShiftDR signal, generated by the state machine in the TAP controller, selects the Shift In data path. The ClockDR signal, also generated by the state machine, clocks the data into the CAP flip-flop. The state machine remains in the Shift-DR state for as many cycles as are needed to completely load the boundary scan register. Then the state machine transitions through the Exit1-DR state, to the Update-DR state. This causes the UPD flip-flop to be loaded. During this operation the Mode input is at 0, so the Preload can be accomplished without interfering with normal operation of the circuit.

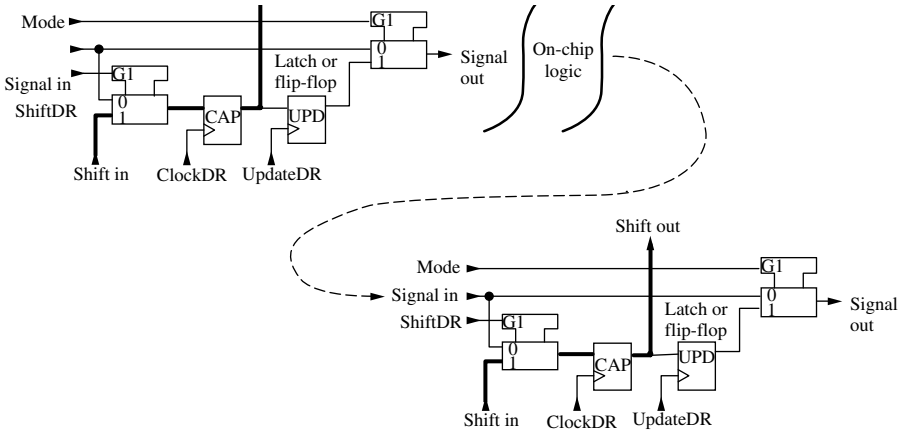


Figure 8.39 Data flow for Preload instruction.

Figure 8.40 illustrates the data flow for the Extest instruction. Recall that the purpose of this instruction is to test interconnect circuitry between IEEE1149.1 compliant chips, as well as clusters of noncompliant chips on the board. The first step in the operation of Extest is to run the Preload instruction in order to load the boundary scan register. The values in the CAP registers are loaded into the UPD register cells upon entering the UpdateDR state of the state machine. Then, the Extest instruction is loaded into the instruction register. The Mode signal changes to a 1, causing the value in the UPD register to appear at Signal Out. In the Capture-DR state, data at the input pins is loaded into the shift-register stage. Then, in the Shift-DR state, results can be shifted out while new data are shifted into the shift-registers. The data shifted out can be inspected to determine if they are correct while the Update-DR state is again entered in order to present new data at the output pins. The process is repeated for as many tests as are needed to completely check the interconnect logic between the chips.

The Sample instruction is used to capture data at the input pins. This is illustrated in Figure 8.41. The Mode input is set to 0, so data at the input pins flow straight through to the internal logic. Data at the output of the internal logic flow through the cell to the output pin. At the same time the data are being captured into the shift-register flip-flops. During debug, these data can be shifted out while the system clock is held inactive. After inspecting the data, the system clock can be single-stepped, and the data can again be captured, shifted out, and inspected.

The discussion presented here is strictly an overview of the material on IEEE1149.1 boundary scan and is intended as a first understanding of its operation. The reader who is required to implement 1149.1 on a working chip should refer to the IEEE standard for much more detailed information of its implementation and operation, including suggested implementations of several more cells, suggested implementations consistent with the LSSD methodology, and data flow descriptions of the

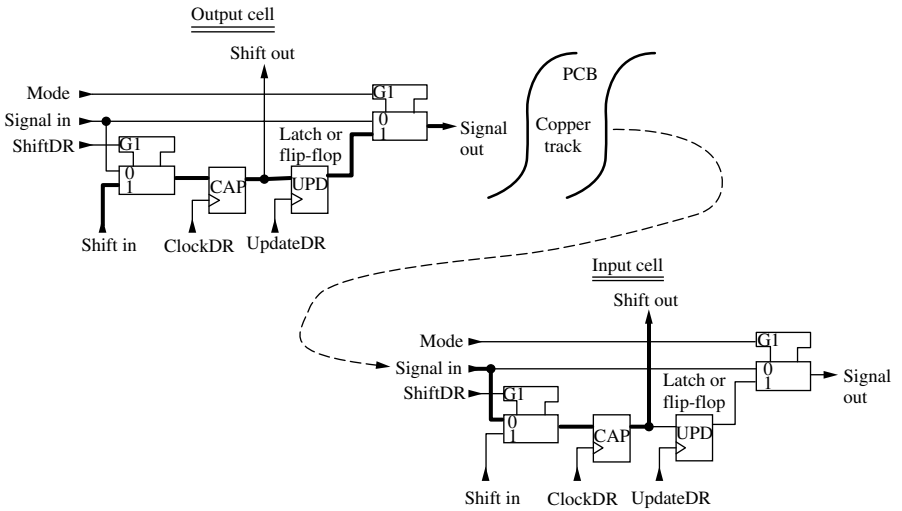


Figure 8.40 Data flow for Extest instruction.

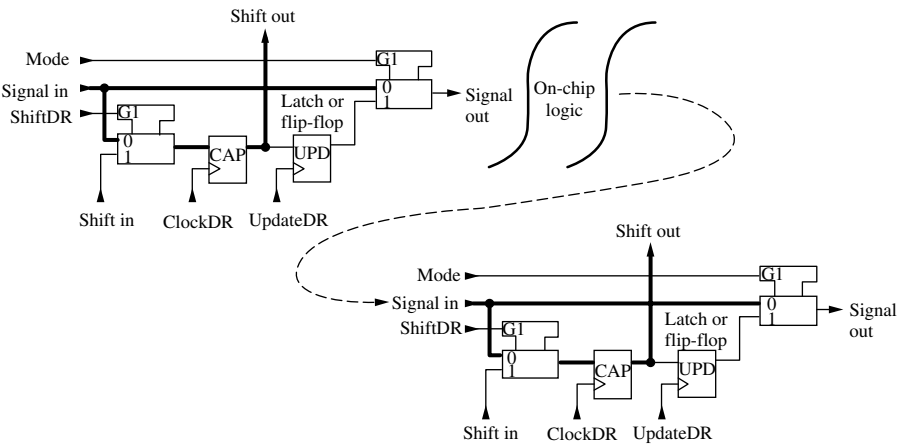


Figure 8.41 Data flow for Sample instruction.

optional instructions. We have not discussed BSDL (boundary scan description language). BSDL is a VHDL subset that is limited to the boundary scan application.[36] Its objective is to serve as an easy to use, machine parsable medium for describing boundary scan implementations. This description can then be used by CAD tools such as synthesis, testability analysis, and test pattern generation.

8.7 SUMMARY

System architects, logic designers, and technologists continue to create ever larger, more complex circuits on increasingly smaller die. The interactions between functions on the chip grow even more rapidly. Sequential ATPG programs do not work on these circuits, and manually generated test pattern generation is not an option. But even if it were possible to solve the complexity issues and create thorough functional tests for these huge chips without having to resort to DFT, the time required to apply the test on the tester would almost always be prohibitive. It was pointed out in Chapter 6 that tester time is expensive. Reducing test cost involves reducing the amount of time required to apply the test. This can be accomplished by (a) employing scan to get high-fault coverage and (b) breaking a scan chain into several smaller chains in order to clock in data and clock out response more quickly.

Testability involves trade-offs. When production volume for an IC is expected to be in the tens of millions of units, the cost of DFT must be examined more carefully than when volume is expected to be low. With sales volume in the millions, the nonrecurring test development cost is amortized over all those parts and cost per unit is likely to be quite low. It may in fact be considerably less than the cost of additional die space, so expending more engineering time to create an efficient test program, or one that uses less die area, can be justified. However, time-to-market concerns and recurring costs, such as the cost of tester time, must still be factored into the equation.

The use of DFT among major vendors of microprocessors is virtually universal. As millions of transistors get integrated onto an IC, DFT is crucial both to the development of effective test programs and to the application of these programs to the IC while the parts are on the tester. It is worth noting that early adaptors of DFT were, for the most part, vertically integrated companies. IC manufacturers are often prone to looking only at the cost of the IC. From that perspective the cost of additional circuitry for test purposes appears as a cost burden. Vertically integrated companies more readily see the benefit of enhanced test results, because the downstream cost of bad ICs is sometimes painfully evident when those results come back from the division or department responsible for stuffing PCBs with those chips (recall the rule-of-ten).

IEEE1149.1 is a DFT methodology that was slow in being adopted by IC vendors. They are prone to looking at it in terms of real estate cost, without considering its value to the system integrator. In fact, it is quite difficult for the IC vendor to justify the presence of boundary scan on a chip. It takes up real estate, becomes another potential source of defects, and contributes no functionality or features that the marketing department can advertise. The PCB board manufacturer, on the other hand, is subject to the "rule-of-ten." He may see an entire PCB discarded, with all of its populated parts, because a fault could not be diagnosed. He has a better appreciation for the value of boundary scan.

When the now legendary floating point design error appeared in an early version of the Pentium chip, it was pointed out by some industry pundits that, for the first time, the vast majority of end users were nontechnical. Whereas in the early days of

the IC revolution most users of high-tech products were likely to be technically inclined, and somewhat forgiving of devices that failed to perform as advertised, the industry has since come to a crossroads where the vast majority of users, being nontechnical and less appreciative of the difficulties inherent in designing and manufacturing state-of-the-art devices, simply want the devices to work. Even when the vendor replaces malfunctioning devices, there is a public relations problem that may have significant adverse effects on the company's reputation (and its bottom line).

Another trade-off that must be assessed is the choice between mean time before failure (MTBF) and mean time to repair (MTTR). The ideal situation is to have systems that never fail, but that may be an unreasonable expectation. It may be preferable to design a more modular system that perhaps invites a slightly shorter duration MTBF but one for which it is easier to detect, diagnose, and correct problems quickly, accurately, and economically in a mass production environment.

The problems of designing testable logic have their parallel in software development, where it was recognized years earlier that complex systems, put together by people with a diverse range of skills and styles, will result in chaos if maintainability is ignored until after the product is designed and developed. In either case, software or hardware design, it is becoming widely recognized and accepted that the designer must ask, before pencil is put to paper on the first design document, "How am I going to diagnose the problems *when* this thing fails?" For the software engineer the answer is structured design. For the logic designer the answer is design-for-testability. Since it is not practical to probe inside a chip after it has been fabricated, testability features must be designed in at the start of a project. This requires that the designer understand testability issues and be able to anticipate testability problems in the design.

At the same time, the project manager must understand cost. It is claimed that, as a rule of thumb,[37] "a 20% increase in area increases chip cost by about 50%." Nevertheless, it can be asserted that

$$\text{Cost(Design + Test)} \leq \text{Cost(Design)} + \text{Cost(Test)}$$

This equation states that product cost is best minimized by viewing design and test as one integral activity rather than disjoint, unrelated activities. When design and test are treated as separate issues, relationships become obscured. Decisions are made on the basis of their impact on the number of I/O pins, amount of board real estate taken up, and number of nanoseconds impact on performance, without considering their impact on production costs such as test development, cost of test application, mean time to repair, scrapped units, rework, retest, and loss of customer good will.

PROBLEMS

8.1 Given a tri-state buffer (bufif1 in Verilog terms), how would you detect a SA1 on the enable input?

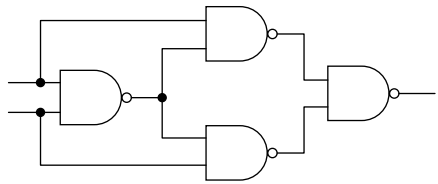


Figure 8.42 NAND version of XOR.

8.2 A circuit has a period of 10 ns. An XOR gate has a delay of 1.5 ns. Using the parity tree of Figure 8.4, what is the maximum number of internal nodes that can be observed in that period?

8.3 Derive the controllability/observability equations for a two-input NAND gate.

8.4 Use the C/O equations for the NAND gate derived in the previous problem to compute controllability/observability numbers for the EXOR circuit in Figure 8.42.

8.5 Compute C/O numbers for the multiplexer circuit in Figure 8.43. Then, create a truth table for the circuit and generate the sets P_1 and P_0 (cf. Section 4.3.1). Intersect these and use the results to generate C/O equations for the multiplexer as a primitive.

8.6 Given a four-input AND gate embedded in a circuit where the CC^0 numbers are (1,1,3,∞) and the CC^1 numbers are (1, 1, 8, ∞) on its inputs, and the combinational observability of its output is 52. Compute the controllability and observability numbers at its output.

8.7 For the delay flip-flop discussed in Section 8.3.1, derive the observability equations for the Data input.

8.8 Derive combinational controllability/observability equations for the circuit described by the truth table given below. Use the equations to compute the controllability/observability numbers.

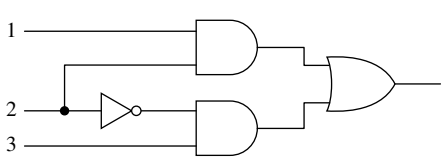


Figure 8.43 XOR—another view.

X_1	X_2	X_3	F
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	0

8.9 Given: a set of two-input AND gates connected as a binary tree with output F. For a tree of depth k in which the inputs are equidistant from the output (same number of nodes between each input and the output), show that

$$CC^1(F) = 2^{k+1} - 1$$
$$CC^0(F) = k + 1$$

8.10 Use the SM8 state machine in Figure 8.44 for Problems 8.10(a) through 8.10(g). Assume the existence of a master reset that initially resets all DFFs to 0.

(a) Use a gate-level, sequential ATPG algorithm of your choice (e.g., EBT, etc.) to find a test for the indicated fault on gate 15.

(b) Create a state transition table, and then write a Verilog description of the circuit (you may find a simulator helpful for this exercise). Map the binary values of $\{Q_2, Q_1, Q_0\}$ onto their decimal equivalents—that is, (0,0,0) $\rightarrow S_0$, (0,0,1) $\rightarrow S_1$, and so on.

(c) Create an S-graph for the state machine. Can you break all cycles by scanning fewer than three flip-flops?

(d) Explain how you might use the results of part (b) to create a guidance file for this state machine (cf. Section 7.10.2).

(e) Convert the circuit by adding scan to the three flip-flops. Create a complete scan test for the indicated fault. Show the sequence of inputs (i.e., the test vectors) that are applied to this circuit in order to detect the fault, and then show the sequence required to scan out and observe the results.

(f) Assume that this state machine is embedded in a circuit and that the flip-flop labeled 17 is to be omitted from the scan path and treated as an X-generator. Identify all the undetectable faults in the cone of 17, and identify all faults that will be only potentially detectable as a result of the X emanating from 17.

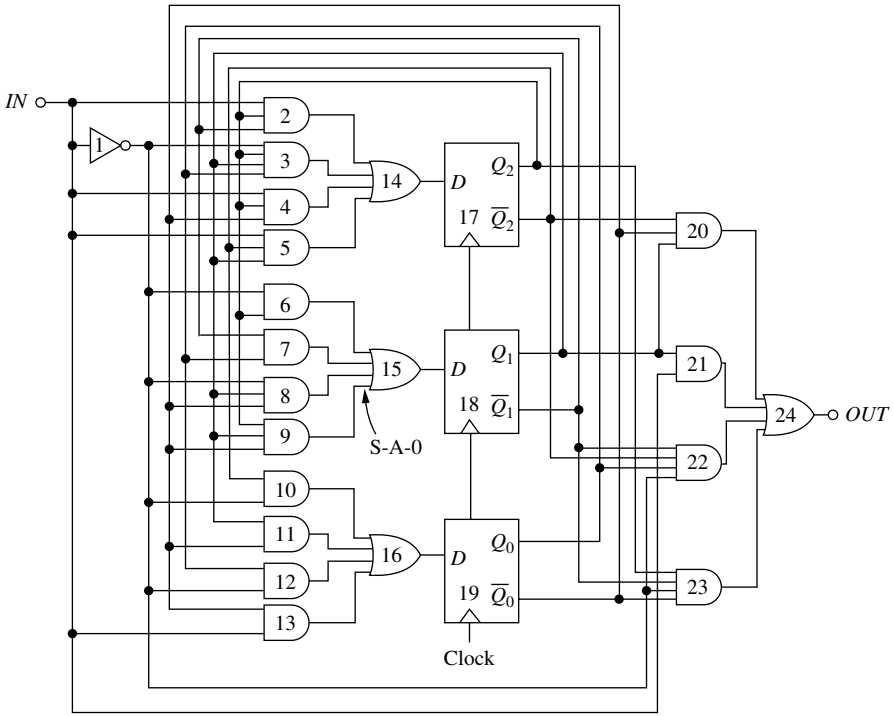


Figure 8.44 The SM8 state machine.

(g) Again, with flip-flop 17 omitted from the scan path, generate a test for the indicated fault using the partial scan path. Show the sequence of inputs applied to the circuit to test for the fault, and show the sequence of outputs required to observe the results.

8.11 A scan path contains 10 scan-flops. Inverters are inserted between the output of each flip-flop in the scan path and the input of the next flip-flop. The Q output of the third scan-flop is SA1. If the scan-flops are reset and then clocked out, what is the resulting output pattern? If, instead, the Q output of the fourth scan-flop is SA1, then what is the resulting output pattern?

8.12 In the testable NAND latch of Figure 8.22, identify the faults that are undetectable when in scan mode.

8.13 A scan circuit has five flip-flops. The first, third and fifth flip-flops are positive-edge triggered. The second and fourth flip-flops are negative-edge triggered. Assuming that you do not take any special steps to address this configuration, describe the sequence of events as you attempt to load the pattern 11001. What are the final contents of the scan chain? Suppose the negative-edge triggered flip-flops are changed by XOR'ing the clock with a

test signal so that all the flip-flops load on the positive edge. Describe the results when loading the scan chain.

8.14 A circuit with 1000 scan-flops has three uniquely identifiable blocks of logic. Block1 has 200 scan-flops and requires 300 scan vectors, block2 had 300 scan-flops and requires 80 scan vectors, and block3 has 500 scan-flops and requires 700 vectors.

(a) If you could not break up the chains, how would you organize them to minimize test time?

(b) If you could break up the longest chain into chains of length 300 and 200, with each requiring 400 scan vectors, how would you organize the chains?

8.15 For the NAND tree of Figure 8.34, assume a device with 200 pins. Assume that the pins are connected in ascending numerical order. Describe the expected waveform when the NAND tree is being exercised. Describe the waveform that results when pin 39 is stuck-at-1.

8.16 Assume the following sequences are applied to the TDI input of a TAP controller. Describe the state transitions that occur in response to the sequences

1111111

11010100011110000

100000001110001101

8.17 Create an S-graph similar to that in Figure 8.33 for a four-stage counter. Can the circuit be broken up for partial scan?

8.18 Documentation can be an important part of a DFT strategy. The circuit in Figure 8.45 uses a 74151 one-of-eight selector (Figure 8.45). Can you identify the function performed by this circuit? Can you guess why it was used?

8.19 Using a DFT circuit, my fault coverage improves from 86.5% to 95.7%. My process yield was 83%. What is the improvement in my AQL?

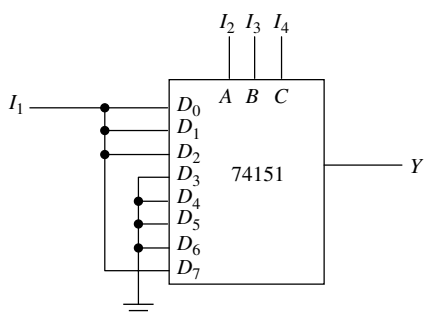


Figure 8.45 One-of-eight selector.

REFERENCES

1. Designing Digital Circuits for Testability, Hewlett-Packard Application Note 210-4, Hewlett Packard, Loveland, CO.
2. Goldstein, L. H., Controllability/Observability Analysis of Digital Circuits, *IEEE Trans. Comput.*, Vol. CAS-26, No. 9, September 1979, pp. 685–693.
3. Powell, T., Software Gauges the Testability of Computer-Designed ICs, *Electron. Des.*, November 24, 1983, pp. 149–154.
4. Fong, J. Y. O., On Functional Controllability and Observability Analysis, *Proc. 1982 Int. Test Conf.*, November 1982, pp. 170–175.
5. Goel, D. K., and R. M. McDermott, An Interactive Testability Analysis Program—ITTAP, *Proc. 19th Des. Autom. Conf.*, 1982, pp. 581–586.
6. Savir, J., Good Controllability and Observability Do Not Guarantee Good Testability, *IEEE Trans. Comput.*, Vol. C-32, No. 12, December 1983, pp. 1198–1200.
7. Agrawal, V. D., and M. R. Mercer, Testability Measures—What Do They Tell Us?, *Proc. Int. Test Conf.* 1982, pp. 391–396.
8. *LASAR User's Manual*, Teradyne Corp., Boston.
9. Levitt, Marc E., Designing UltraSparc for Testability, *IEEE Des. Test*, Vol. 14, No. 1, January–March 1997, pp. 10–17.
10. Ando, H., Testing VLSI with Random Access Scan, *Dig. CompCon.1980*, February 1980, pp. 50–52.
11. Maling, K., and E. L. Allen, A Computer Organization and Programming System for Automated Maintenance, *IEEE Trans. Electron. Comput.*, Vol. EC-12, December 1963, pp. 887–895.
12. Carter, W. C. et al., Design of Serviceability Features for the IBM System/360, *IBM J. Res. Dev.*, Vol. 8, April 1964, pp. 115–126.
13. Hirtle, A. C. et al., Data Processing System Having Auxiliary Register Storage, U.S. Patent No. 3,582,902, filed December 30, 1968.
14. Williams, M. J. Y., and J. B. Angell, Enhancing Testability of Large-Scale Integrated Circuits via Test Points and Additional Logic, *IEEE Trans. Comput.*, Vol. C-22, No. 1, January 1973, pp. 46–60.
15. Eichelberger, E. B., and T. W. Williams, A Logic Design Structure for LSI Testability, *Proc. 14th Des. Autom. Conf.*, June 1977, pp. 462–468.
16. Bottorff, P. S. et al., Test Generation for Large Logic Networks, *Proc. 14th Des. Autom. Conf.*, June 1977, pp. 479–485.
17. Godoy, H. C. et al., Automatic Checking of Logic Design Structures for Compliance with Testability Ground Rules, *Proc. 14th Des. Autom. Conf.*, June 1977, pp. 469–478.
18. Cheung, B., and L. T. Wang, The Seven Deadly Sins of Scan-Based Designs, *Integrated Syst. Des.*, August 1997, pp. 50–56.
19. Yohannes, Paul, Useful Design-for-Test Practices, *ISD Mag.*, September 2000, pp. 58–66.
20. Jaramillo, K., and S. Meiyappan, 10 Tips for Successful Scan Design: Part One, *EDN Mag.,* February 17, 2000, pp. 67–75.
21. Jaramillo, K., and S. Meiyappan, 10 Tips for Successful Scan Design: Part Two, *EDN Mag.*, February 17, 2000, pp. 77–90.

22. Narayanan, S. et al., Optimal Configuring of Multiple Scan Chains, *IEEE Trans. Comput.*, Vol. 42, No. 9, September 1993, pp. 1121–1131.
23. Anderson, T. L., and C. K. Allsup, Incorporating Partial Scan, *ASIC & EDA*, October 1994, pp. 23–32.
24. Stewart, J. H., Future Testing of Large LSI Circuit Cards, *Proc. 1977 Cherry Hill Test Conf.*, October 1977, pp. 6–17.
25. Trischler, Erwin, Incomplete Scan Path with an Automatic Test Generation Methodology, *Proc. Int. Test Conf.*, 1980, pp. 153–162.
26. Agrawal, V. et al., Designing Circuits with Partial Scan, *IEEE Des. Test Comput.*, 1988, pp. 8–15.
27. Morley, S. P., and R. A. Marlett, Selectable Length Partial Scan: A Method to Reduce Vector Length, *Proc. Int. Test Conf.*, 1991, pp. 385–392.
28. Cheng, K. T., and V. D. Agrawal, A Partial Scan Method for Sequential Circuits with Feedback, *IEEE Trans. Comput.*, April 1990, pp. 544–548.
29. Chen, C. et al., Layout Driven Selecting and Chaining of Partial Scan Flip-Flops, *Proc. Des. Auto. Conf.*, 1996.
30. Chickermane, V., and J. H. Patel, An Optimization Based Approach to the Partial Scan Design Problem, *Proc. Int. Test Conf.*, 1990, pp. 377–386.
31. Chickermane, V., and J. H. Patel, A Fault Oriented Partial Scan Design Approach, *Proc. Int. Test Conf.*, 1991, pp. 400–403.
32. Hudli, R. V., and S. C. Seth, Testability Analysis of Synchronous Sequential Circuits Based on Structural Data, *Proc. Int. Test. Conf.*, 1989, pp. 364–372.
33. Hewlett-Packard Co., Section 1.1.5, The Manufacturing Fault Spectrum and Boundary Scan, *Boundary-Scan Tutorial*, Rev. G, 1990, pp. 1–13.
34. Dody, G., Troubleshooting BGAs, *SMT: The Magazine for Electronics Assembly*, July 1999, pp. 44–50.
35. IEEE, *IEEE Standard Test Access Port and Boundary Scan Architecture*, IEEE Standards Board, New York, IEEE Standard 1149.1-1990, May 1990.
36. Parker, K. P., *The Boundary-Scan Handbook*, Kluwer Academic Publishers, Boston, 1992.
37. Walker, Martin G., Modeling the Wiring of Deep Submicron ICs, IEEE Spectrum, March 2000, p. 67.

CHAPTER 9

Built-In Self-Test

9.1 INTRODUCTION

Numerous ATPG algorithms and heuristics have been developed over the years to test digital logic circuits. Some of these methods can trace their origins back to the very beginnings of the digital logic era. Unfortunately, they have proven inadequate to the task. Despite many novel and interesting schemes designed to attack test problems in digital circuits, circuit complexity and the sheer number of logic devices on a die continue to outstrip the test schemes that have been developed, and there does not appear to be an end in sight, as levels of circuit integration continue to grow unabated.

New methods for testing and verifying physical integrity are being researched and developed. Where once the need for concessions to testability was questioned, now, if there is any debate at all, it usually centers on what kind of testability enhancements should be employed. However, even with design-for-testability (DFT) guidelines, difficulties remain. Circuits continue to grow in both size and complexity. When operating at higher clock rates and lower voltages, circuits are susceptible to performance errors that are not well-modeled by stuck-at faults. As a result, there is a growing concern for the effectiveness as well as the cost of developing and applying test programs.

Test problems are compounded by the fact that there is a growing need to develop test strategies both for circuits designed in-house and for intellectual property (IP) acquired from outside vendors. The IP, often called core modules or soft cores, can range from simple functions to complex microprocessors. For test engineers, the problem is compounded by the fact that they must frequently develop effective test strategies for devices when description of internal structure is unavailable.

There is a growing need to develop improved test methods for use at customer sites where test equipment is not readily accessible or where the environment cannot be readily duplicated, as in military avionics subject to high gravity stresses while in operation. This has led to the concept of built-in self-test (BIST), wherein

Digital Logic Testing and Simulation, *Second Edition*, by Alexander Miczo
ISBN 0-471-43995-9 Copyright © 2003 John Wiley & Sons, Inc.

test circuits are placed directly within the product being designed. Since they are closer to the functions they must test, they have greater controllability and observability. They can exercise the device in its normal operating environment, at its intended operating speed, and can therefore detect failures that occur only in the field. Another form of BIST, error detection and correction (EDAC) circuits, goes a step further. EDAC circuits, used in communications, not only detect transmission errors in noisy channels, but also correct many of the errors while the equipment is operating.

This chapter begins with a brief look at the benefits of BIST. Then, circuits for creating stimuli and monitoring response are examined. The mathematical foundation underlying these circuits will be discussed, followed by a discussion of the effectiveness of BIST. Then some case studies are presented describing how BIST has been incorporated into some complex designs. Test controllers, ranging from fairly elementary to quite complex, will be examined next. Following that, circuit partitioning will be examined. Done effectively, it affords an opportunity to break a problem into subproblems, each of which may be easier to solve and may allow the user to select the best tool for each subcircuit or unit in a system. Finally, fault tolerance is examined.

9.2 BENEFITS OF BIST

Before looking in detail at BIST, it is instructive to consider the motives of design teams that have used it in order to understand what benefits can be derived from its implementation. Bear in mind that there is a trade-off between the perceived benefits and the cost of the additional silicon needed to accommodate the circuitry required for BIST. However, when a design team has already committed to scan as a DFT approach, the additional overhead for BIST may be quite small. BIST requires an understanding of test strategies and goals by design engineers, or a close working relationship between design and test engineers. Like DFT, it imposes a discipline on the logic designer. However, this discipline may be a positive factor, helping to create designs that are easier to diagnose and debug.

A major argument for the use of BIST is the reduced dependence on expensive testers. Modern-day testers represent a major investment. To the extent that this investment can be reduced or eliminated, BIST grows in attractiveness as an alternative approach to test. It is not even necessary to completely eliminate testers from the manufacturing flow to economically justify BIST. If the duration of a test can be reduced by generating stimuli and computing response on-chip, it becomes possible to achieve the same throughput with fewer, and possibly less expensive, testers. Furthermore, if a new, faster version of a die is released, the BIST circuits also benefit from that performance enhancement, with the result that the test may complete in less time.

One of the problems associated with the testing of ICs is the interface between the tester and the IC. Cables, contact pins, and probe cards all require careful attention because of the capacitance, resistance, and inductance introduced by these

devices, as well as the risk of failure to make contact with the pins of the device under test (DUT), possibly resulting in false rejects. These interface devices not only represent possible technical problems, they can also represent a significant incremental equipment cost. BIST can eliminate or significantly reduce these costs.

Many circuits employ memory in the form of RAM, ROM, register banks, and scratch pads. These are often quite difficult to access from the I/O pins of an IC; sometimes quite elaborate sequences are needed to drive the circuit into the right state before it is possible to apply stimuli to these embedded memories. BIST can directly access these memories, and a BIST controller can often be shared by some or all of the embedded memories.

Test data generation and management can be very costly. It includes the cost of creating, storing, and otherwise managing test patterns, response data, and any diagnostic data needed to assist in the diagnosis of defects. Consider the amount of data required to support a scan-based test. For simplicity, assume the presence of a single scan path with 10,000 flip-flops and assume that 500 scan vectors are applied to the circuit. The 500 test vectors will require 5,000,000 bits of storage (assuming 1 bit for each input, that is, only 0 and 1 values allowed). Given that a 10,000-bit response vector is scanned out, a total of 10,000,000 bits must be managed for the scan test. This does not represent a particularly large circuit, and the test data may have to be replicated for several revision levels of the product, so the logistics involved may become extremely costly.

BIST can help to substantially reduce this data management problem. When using BIST to test a circuit, it may be that the only input stimulus required is a reset that puts the circuit into test mode and forces a seed value in a pseudo-random pattern generator (PRG). Then, if a tester is controlling the self-test, a predetermined number of clocks are applied to the circuit and a response, called a signature, is read out and compared to the expected signature. If the signature is compressed into a 32-bit signature, many such signatures can be stored in a small amount of storage.

Another advantage of BIST is that many thousands of pseudo-random vectors can be applied in BIST mode in the time that it takes to load a scan path a few hundred times. The test vectors come from the PRG, so there is no storage requirement for test vectors. It should also be noted that loading the scan chain(s) for every vector can be time-consuming, implying tester cost, in contrast to BIST where a seed value is loaded and then the PRG immediately starts generating and applying a series of test vectors on every clock. A further benefit of BIST is the ability to run at speed, which improves the likelihood of detecting delay errors.

Some published case studies of design projects that used BIST stress the importance of being able to use BIST during field testing.[1] One of the design practices that supports field test is the use of flip-flops at the boundaries of the IC.[2] These flip-flops can help to isolate an IC from other logic on the PCB, making it possible to test the IC independent of that other logic. This makes it possible to diagnose and repair PCBs that otherwise might be scrapped because a bad IC could not be accurately identified.

There is a growing use of BIST in personal computers (PCs). The Desktop Management Task Force (DMTF) is establishing standards to promote the use of BIST

for PCs.[3] If a product adheres to the standard, then test programs can be loaded into memory and executed from the vendor's maintenance depot, assuming that the PC has a modem and is not totally dead, so a field engineer may already have a good idea what problems exist before responding to a service request.

9.3 THE BASIC SELF-TEST PARADIGM

The built-in-self-test approach, in its simplest form, is illustrated in Figure 9.1 Stimuli are created by a *pseudo-random generator (PRG)*. These are applied to a combinational logic block, and the results are captured in a *signature analyzer*, or *test response compactor (TRC)*. The PRG could be something as simple as an n-stage counter, if the intent is to apply all possible input combinations to the combinational logic block. However, for large values of n ($n \geq 20$), this becomes impractical. It is also unnecessary in most cases, as we shall see. A *linear-feedback shift register (LFSR)* generates a reasonably random set of patterns that, for most applications, provides adequate coverage of the combinational logic with just a few hundred patterns. These pseudo-random patterns may also be more effective than patterns generated by a counter for detecting CMOS stuck-open faults.

The TRC captures responses emanating from the combinational logic and compresses them into a vector, called a signature, by performing a transformation on the bit stream. This signature is compared to an expected signature to determine if the logic responded correctly to the applied stimuli. There are any number of ways to generate a signature from a bit stream. It is possible, when sampling the bit stream, to count 1s. Each individual output from the logic could be directed to an XOR, essentially a series of one-bit parity checkers. It is also possible to count transitions, with the data stream clocking a counter.

Another approach adds the response at the end of each clock period to a running sum to create a checksum. The checksum has uneven error detection capability. If a double error occurs, and both bits occur in the low-order column, the low-order bit is unchanged but, because of the carry, the next-higher-order bit will be complemented and the error will be detected. If the same double bit error occurs in the high-order bit position, and if the carry is overlooked, which may be the case with checksums, the double error will go undetected.

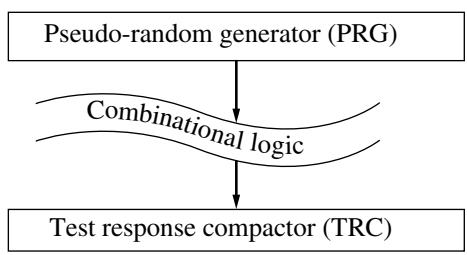


Figure 9.1 Basic self-test configuration.

In fact, if there is a stuck-at-e condition, $e \in \{0,1\}$, affecting the entire high-order bit stream, either at the sending or receiving end, there is only a 50% chance that it will be detected by a checksum that ignores carries. Triple errors can also go undetected. A double error in the next-to-high-order position, occurring together with a single bit error in the high-order position, will again cause a carry out but have no effect on the checksum. In general, any multiple error that sums to zero, with a carry out of the checksum adder, will go undetected.

Example Given a set of n 8-bit words for which a checksum is to be computed, assume that the leftmost columns of four of the words are corrupted by errors e_1 through e_4, as shown.

e_1		0	0	1	0	0	0	0	0
e_2		0	1	0	0	0	0	0	0
e_3		1	0	0	0	0	0	0	0
e_4		0	0	1	0	0	0	0	0
	1	0	0	0	0	0	0	0	0

The errors sum to zero, hence they will go undetected if the carry is ignored. Note that the leftmost column has odd parity, so if the input to the checksum circuit was stuck-at-1, the same erroneous result would occur. ■■

A more commonly used constuct for creating signatures is the multiple-input shift register (MISR), also sometimes called a multiple-input signature register. The MISR and the PRG are based on the linear feedback shift register (LFSR). Before looking at implementation details, some theoretical concepts will be examined.

9.3.1 A Mathematical Basis for Self-Test

This section provides a mathematical foundation for the PRG and MISR constructs. The mathematics presented here will provide some insight into why some circuits are effective and others ineffective, and will also serve as a basis for the error-correcting codes presented in Chapter 10.

We start with the definition of a group. A *group* G is a set of elements and a binary operator * such that

1. $a, b, \in G$ implies that $a * b \in G$ (closure)
2. $a, b, c \in G$ implies that $(a * b) * c = a * (b * c)$ (associativity)
3. There exists $e \in G$ such that $a * e = e * a$ for all $a \in G$ (identity)
4. For every $a \in G$, there exists
   $a^{-1} \in G$ such that $a * a^{-1} = a^{-1} * a = e$ (inverse)

A group is *commutative*, also called *abelian*, if for every $a, b \in G$ we have $a * b = b * a$.

Example The set $I = \{\ldots, -2, -1, 0, 1, 2, \ldots\}$ and the operator * form a group when * represents the usual addition (+) operation. ■■

Example The set $S = \{S_i \mid 0 \le i \le 3\}$ of squares is defined as follows: S_0 has a notch in the upper left corner and S_i represents a clockwise rotation of S_0 by $i \times 90$ degrees. A rotation operator R is defined such that $S_i R S_j = S_k$, where $k = i + j$ (modulo 3). The set S and the operator R satisfy the definition of a group. The element S_k is simply the result of S_i and S_j applied in succession. ■■

Given a group G with n elements and identity 1, the number of elements in G is called the *order of G*. The order of an element $g \in G$ is the smallest integer e such that $g^e = 1$. It can be shown that e divides n.

A *ring* R is a set of elements on which two binary operators, + and ×, are defined and satisfy the following properties:

1. The set R is an *Abelian group* under +
2. $a, b \in R$ implies that $a \times b \in$ R
3. $a, b, c \in R$ implies that $(a \times b) \times c = a \times (b \times c)$
4. $a, b, c \in R$ implies that
   $a \times (b + c) = a \times b + a \times c$
   $(b + c) \times a = b \times a + c \times a$

If the set R also satisfies

5. $a \times b = b \times a$

then it is a *commutative ring*.

Example The set of even integers is a commutative ring. ■■

A commutative ring that has a multiplicative identity and a multiplicative inverse for every nonzero element is called a *field*.

Example The set of elements {0,1} in which + is the exclusive-OR and × is the AND operation satisfies all the requirements for a field and defines the Galois field GF(2). ■■

Given a set of elements V and a field F, with u, v and $w \in V$ and a, b, c, $d \in F$, then V is a *vector space over F* if it satisfies the following:

1. The product $c \cdot v$ is defined, and $c \cdot v \in V$
2. V is an Abelian group under addition
3. $c \cdot (u + v) = c \cdot u + c \cdot v$
4. $(c + d) \cdot v = c \cdot v + d \cdot v$

5. $(c \cdot d) \cdot v = c \cdot (d \cdot v)$
6. $1 \cdot v = v$ where 1 is the multiplicative identity in F

The field F is called the *coefficient field*. It is GF(2) in this text, but GF(p), for any prime number p, is also a field. The vector space V defined above is a *linear associative algebra over F* if it also satisfies the following:

7. The product $u \cdot v$ is defined and $u \cdot v \in V$
8. $(u \cdot v) \cdot w = u \cdot (v \cdot w)$
9. $u \cdot (c \cdot v + d \cdot w) = c \cdot u \cdot v + d \cdot u \cdot w$
   $(c \cdot v + d \cdot \mathrm{w}) \cdot u = c \cdot v \cdot u + d \cdot w \cdot u$

The *Euclidean division algorithm* states that for every pair of polynomials $S(x)$ and $D(x)$, there is a unique pair of polynomials $Q(x)$ and $R(x)$ such that

$$S(x) = D(x) \cdot Q(x) + R(x)$$

and the degree of $R(x)$ is less than the degree of $D(x)$. The polynomial $Q(x)$ is called the *quotient* and $R(x)$ is called the *remainder*. We say that $S(x)$ is equal to $R(x)$ modulo $D(x)$. The set of all polynomials equal to $R(x)$ modulo $D(x)$ forms a *residue class* represented by $R(x)$. If $S(a) = 0$, then a is called a *root* of $S(x)$.

A natural correspondence exists between vector n-tuples in an algebra and polynomials modulo $G(x)$ of degree n. The elements $a_0, a_1, \ldots, a_{n-1}$ of a vector v correspond to the coefficients of the polynomial

$$b_0 + b_1 g + b_2 g^2 + \cdots + b_{n-1} g^{n-1}$$

The sum of two n-tuples corresponds to the sum of two polynomials and scalar multiplication of n-tuples and polynomials is also similar. In fact, except for multiplication, they are just different ways of representing the algebra. If $F(x) = x^n - 1$, then the vector product has its correspondence in polynomial multiplication. When multiplying two polynomials, modulo $F(x)$, the coefficient of the ith term is

$$c_i = a_0 b_i + a_1 b_{i-1} + \cdots + a_i b_0 + a_{i+1} b_{n-1} + a_{i+2} b_{n-2} + \cdots + a_{n-1} b_{i+1}$$

Since $x^n - 1 = 0$, it follows that $x^{n+j} = x^j$, and the ith term of the polynomial product corresponds to the inner, or dot, product of vector a and vector b when the elements of b are in reverse order and shifted circularly $i + 1$ positions to the right.

Theorem 9.1 The residue classes of polynomials modulo a polynomial $f(x)$ of degree n form a *commutative linear algebra* of dimension n over the coefficient field.

A polynomial of degree n that is not divisible by any polynomial of degree less than n but greater than 0 is called *irreducible*.

Theorem 9.2 Let $p(x)$ be a polynomial with coefficients in a field F. If $p(x)$ is irreducible in F, then the algebra of polynomials over F modulo $p(x)$ is a field.

The field of numbers 0, 1, …, $q-1$ is called a *ground field*. The field formed by taking polynomials over a field GF(q) modulo an irrreducible polynomial of degree m is called an *extension field*; it defines the field GF(q^m). If $z = \{x\}$ is the residue class, then $p(z) = 0$ modulo $p(x)$, therefore $\{x\}$ is a root of $p(x)$.

If $q = p$, where p is a prime number, then, by Theorem 9.2, the field GF(p^m), modulo an irreducible polynomial $p(x)$ of degree m, is a vector space of dimension m over GF(p) and thus has p^m elements. Every finite field is isomorphic to some Galois field GF(p^m).

Theorem 9.3 Let $q = p^m$, then the polynomial $x^{q-1}-1$ has as roots all the $p^m - 1$ nonzero elements of GF(p^m).

Proof The elements form a multiplicative group. So, the order of each element of the group must divide the order of the group. Therefore, each of the $p^m - 1$ elements is a root of the polynomial $x^q - 1$. But the polynomial $x^q - 1$ has, at most, $p^m - 1$ roots. Hence, all the nonzero elements of GF(p^m) are roots of $x^q - 1$.

If $z \in$ GF(p^m) has order $p^m - 1$, then it is *primitive*.

Theorem 9.4 Every Galois field GF(p^m) has a primitive element; that is, the multiplicative group of GF(p^m) is cyclic.

Example GF(2^4) can be formed modulo $F(x) = x^4 + x^3 + 1$. Let $z = \{x\}$ denote the residue class x; that is, z represents the set of all polynomials that have remainder x when divided by $F(x)$. Since $F(x) = 0$ modulo $F(x)$, x is a root of $F(x)$. Furthermore, x is of order 15. If the powers of x are divided by $F(x)$, the first six division operations yield the following remainders:

$$
\begin{aligned}
x^0 &= 1 && \text{modulo } F(x) = (1,0,0,0)\\
x^1 &= x && \text{modulo } F(x) = (0,1,0,0)\\
x^2 &= x^2 && \text{modulo } F(x) = (0,0,1,0)\\
x^3 &= x^3 && \text{modulo } F(x) = (0,0,0,1)\\
x^4 &= 1 + x^3 && \text{modulo } F(x) = (1,0,0,1)\\
x^5 &= 1 + x + x^3 && \text{modulo } F(x) = (1,1,0,1)
\end{aligned}
$$

■■

The interested reader can complete the table by dividing each power of x by $F(x)$. With careful calculations, the reader should be able to confirm that $x^{15} = 1$ modulo $F(x)$ but that no lower power of x equals 1 modulo $F(x)$. Furthermore, when dividing x^i by $F(x)$, the coefficients are cyclic; that is, if the polynomials are represented in vector form, then each vector will appear in all of its cyclic shifts.

9.3.2 Implementing the LFSR

The LFSR is a basic building block of BIST. A simple n-stage counter can generate 2^n unique input vectors, but the high-order bit would not change until half the stimuli had been created, and it would not change again until the counter returned to its starting value. By contrast, the LFSR can generate pseudo-random sequences and it can be used to create signatures. When used to generate stimuli, the stimuli can be obtained serially, from either the high- or low-order stage of the LFSR, or stimuli can be acquired from all of the stages in parallel. The theory on LFSRs presented in the previous section allows for LFSRs of any degree. However, the polynomials that tend to get the most attention are those that correspond to standard data bus widths—for example, 16, 32, and so on. The LFSR is made up of delays (flip-flops or latches), XORs, and feedback lines. From a mathematical perspective, XORs are modulo 2 adders in GF(2). The circuit in Figure 9.2 implements the LFSR defined by the equation

$$p(x) = x^{16} + x^9 + x^7 + x^4 + 1$$

If the LFSR has no inputs and is seeded with a nonzero starting value—for example, by a reset that forces one or more of the flip-flops to assume nonzero initial values—then the circuit becomes an autonomous LFSR (ALFSR). If the connections correspond to a primitive polynomial, the LFSR is capable of generating a nonrepeating sequence of length 2^n, where n is the number of stages. With the input signal In shown in Figure 9.2 the circuit functions as a TRC.

If the incoming binary message stream is represented as a polynomial $m(x)$ of degree n, then the circuit in Figure 9.2 performs a division

$$m(x) = q(x) \cdot p(x) + r(x)$$

The output is 0 until the 16th shift. After n shifts ($n \geq 16$) the output of the LFSR is a quotient $q(x)$, of degree $n - 16$. The contents of the delay elements, called the signature, are the remainder. If an error appears in the message stream, such that the incoming stream is now $m(x) + e(x)$, then

$$m(x) + e(x) = q'(x) \cdot p(x) + r'(x)$$

and

$$r'(x) = r(x)$$

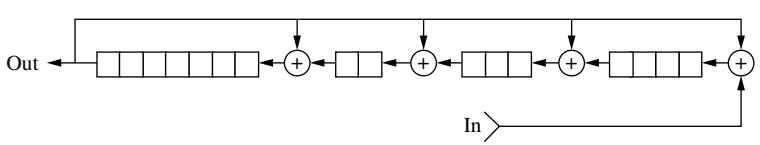


Figure 9.2 Linear feedback shift register.

if and only if $e(x)$ is divisible by $p(x)$. Therefore, if the error polynomial is not divisible by $p(x)$, the signature in the delay elements will reveal the presence of the error.

The LFSR in Figure 9.3 is a variation of the LFSR in Figure 9.2. It generates the same quotient as the LFSR in Figure 9.2, but does not generally create the same remainder. Regardless of which implementation is employed, the following theorem holds:[4]

Theorem 9.5 Let $s(x)$ be the signature generated for input $m(x)$ using the polynomial $p(x)$ as a divisor. For an error polynomial $e(x)$, m(x) and $m(x) + e(x)$ have the same signature if and only if $e(x)$ is a multiple of $p(x)$.

One of the interesting properties of LFSRs is the following:[5]

Theorem 9.6 An LFSR based on any polynomial with two or more nonzero coefficients detects all single-bit errors.

Binary bit streams with 2 bits in error can escape detection. One such example occurs if

$$p(x) = x^4 + x^3 + x + 1$$

and

$$e(x) = (x^6 + 1) \cdot x^n$$

It can also be shown that, if the polynomial has an even number of terms, then it will detect all odd numbers of errors. In addition, all single bursts of length less than the degree of the polynomial will be detected.

9.3.3 The Multiple Input Signature Register (MISR)

The signature generators in Figures 9.2 and 9.3 accumulate signatures by serially shifting in a bit at a time. However, that is impractical for circuits where it is desired to compact signatures while a device is running in its normal functional mode. A more practical configuration is shown in Figure 9.4. Two functional registers serve a

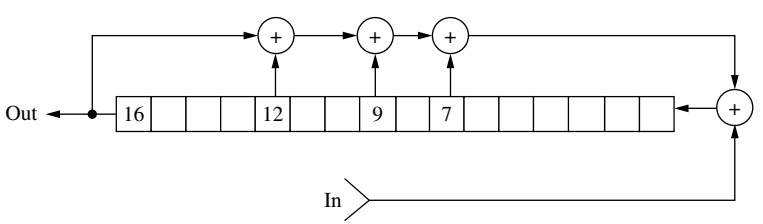


Figure 9.3 Equivalent LFSR.

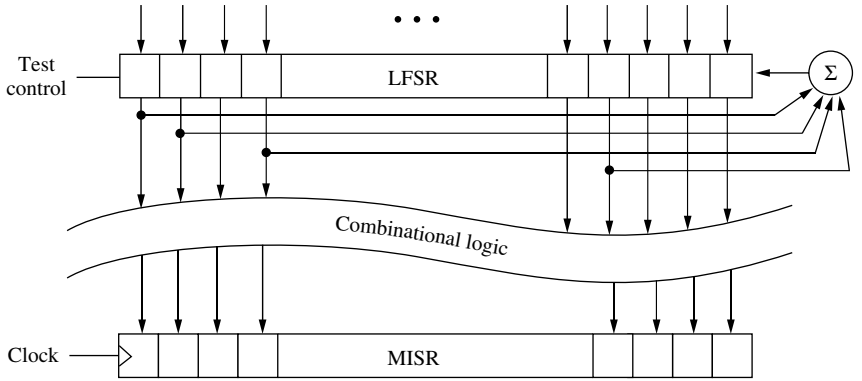


Figure 9.4 Test configuration using maximal LFSR and MISR.

dual purpose. When in self-test mode, one acts as an LFSR and generates as many as $2^m - 1$ consecutive distinct m-bit values that are simultaneously taken from m flip-flops. A second functional register is connected to the output of the combinational logic. It compacts the stimuli to create a signature. A test controller is used to put the register into test mode, seed it with an initial value, and control the number of pseudo-random patterns that are to be applied to the combinational logic.

The MISR is a feedback shift register that forms a signature on n inputs in parallel. After an n-bit word is added, modulo 2, to the contents of the register, the result is shifted one position before the next word is added. The MISR can be augmented with combinational logic in such a way that the generated signature is identical to that obtained with serial compession.[6] The equations are computed for a given LFSR implementation by assuming an initial value c_i in each register bit position r_i, serially shifting in a vector (b_0, b_1, ..., b_{n-1}), and computing the new contents (r_1, r_2, ..., r_n) of the register following each clock. After n clocks the contents of each r_i are specified in terms of the original register contents (c_1, c_2, ..., c_n) and the new data that were shifted in. These new contents of the r_i define the combinational logic required for the MISR to duplicate the signature in the corresponding LFSR.

Example A register corresponding to the polynomial $p(x) = x^4 + x^2 + x + 1$ will be used. The register is shown in equivalent form in Figure 9.5. Assume initially that flip-flop r_i contains c_i. The data bits enter serially, starting with bit b_0. The contents of the flip-flops are shown for the first two shifts. After two more shifts and also making extensive use of the fact that $a \oplus a = 0$ and $a \oplus 0 = a$, the contents of the flip flops are

$$r_1 = c_1 \oplus c_2 \oplus c_3 \oplus b_0$$
$$r_2 = c_2 \oplus c_3 \oplus c_4 \oplus b_1$$
$$r_3 = c_1 \oplus c_2 \oplus b_0 \oplus b_2$$
$$r_4 = c_1 \oplus b_0 \oplus b_1 \oplus b_3$$

■■

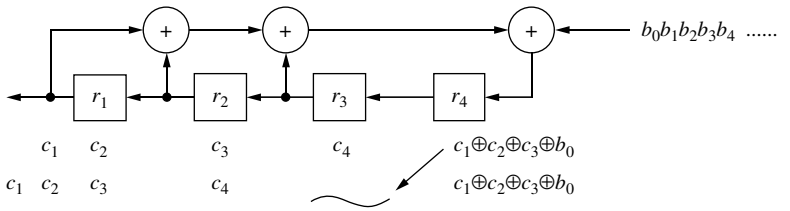


Figure 9.5 Fourth-degree LFSR.

For the purpose of generating effective signatures, it is not necessary that parallel data compression generate a signature that matches the signature generated using serial data compression. What is of interest is the probability of detecting an error. As it turns out, the MISR has the same error detection capability as the serial LFSR when they have an identical number of stages. In the discussion that follows, the equivalence of the error detection capability is informally demonstrated.

Using serial data compression and an LFSR of degree r and also given an input stream of k bits, $k \geq r$, there are $2^{k-r} - 1$ undetectable errors since there are $2^{k-r} - 1$ nonzero multiples of $p(x)$ of degree less than k that have a remainder $r(x) = 0$.

When analyzing parallel data compression, it is convenient to use the linearity property that makes it possible to ignore message bits in the incoming data stream and focus on the error bits. When clocking the first word into the register, any error bit(s) can immediately be detected. Hence, as in the serial case, when $k = r$ there are no undetectable errors. However, if there is an error pattern in the first word, then the second word clocked in is added (modulo 2) to a shifted version of the first word. Therefore, if the second word has an error pattern that matches an error pattern in the shifted version of the first word, it will cancel out the error pattern contained in the register, and the composite error contained in the first and second words will go undetected.

For a register of length r, there are $2^r - 1$ error patterns possible in the first word, each of which, after shifting, could be canceled by an error pattern in the second word. When compressing n words, there are $2^{(n-1)r} - 1$ error patterns in the first $n - 1$ words. Each of these error patterns could go undetected if there is an error pattern in the nth word that matches the shifted version of the error pattern in the register after the first $n - 1$ words. So, after n words, there are $2^{(n-1)r} - 1$ undetectable error patterns. Note that an error pattern in the first $n - 1$ words that sums to zero is vacuously canceled by the all-zero "error" in the nth word. The number of errors matches the number of undetectable errors in a serial stream of length $n \cdot r$ being processed by a register of length r.

Example Using the LFSR in Figure 9.3, if an error pattern $e_1 = 00000000\ 01000000$ is superimposed on the message bits, then after one shift of the register the error pattern becomes $e_2 = 0000000010000001$. Therefore, if the second word contains an error pattern matching e_2, it will cancel the error in the first word, causing the error to go undetected. ■■

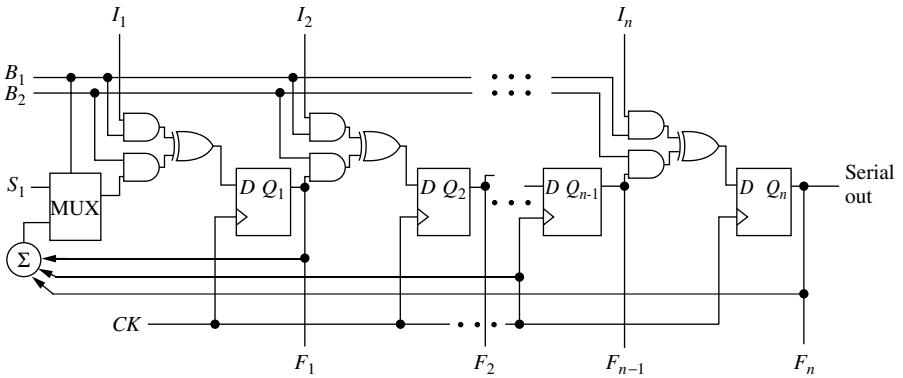


Figure 9.6 BILBO.

9.3.4 The BILBO

The circuit in Figure 9.4 adds logic to a functional register to permit dual-purpose operation: normal functional mode and test response compaction. A more general solution is the built-in logic block observer (BILBO).[7] The BILBO, shown in Figure 9.6, has four modes of operation: When B_1, $B_2 = 0,0$, it is reset. When B_1, $B_2 = 1,0$, it can be loaded in parallel and used as a conventional register. When B_1, $B_2 = 0,1$, it can be loaded serially and incorporated as part of a serial scan path. When B_1, $B_2 = 1,1$, it can be used as an MISR to sum the incoming data $I_1 - I_n$, or, if the data are held fixed, it can create pseudo-random sequences of outputs.

There are a number of ways in which the BILBO can be used. One approach is to convert registers connected to a bus into BILBOs. Then, as depicted in Figure 9.7, either BILBO1 can generate stimuli for combinational logic while BILBO2 generates signatures, or BILBO1 can be configured to generate signatures on the contents of the bus. In that case, the stimulus generator can be another BILBO or a ROM whose contents are being read out onto the bus. After the signature has been generated, it can be scanned out by putting the BILBOs into serial scan mode. Then, assuming that the results are satisfactory, the BILBOs are restored to operational mode.

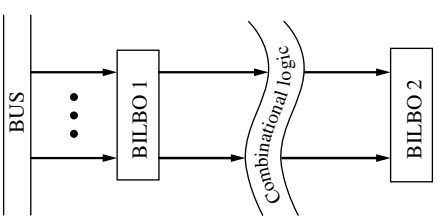


Figure 9.7 BILBO used to test circuit.

In a complex system employing several functional units, there may be several BILBOs and it becomes necessary to control and exercise them in correct order. Hence, a controller must be provided to ensure orderly self-test in which the correct units are generating stimuli and forming signatures, scanning out contents and comparing signatures to verify their correctness.

9.4 RANDOM PATTERN EFFECTIVENESS

Signature analysis compresses long bit strings into short signatures. Nevertheless, it is important to bear in mind that the quality of a test is still dependent on the stimuli used to sensitize and detect faults. In order for a fault to be detected, the stimuli must induce that fault to create an error signal in the output stream.

9.4.1 Determining Coverage

The ideal test is an exhaustive test—that is, one in which all possible combinations are applied to the combinational logic accessed by a scan path. This is all the more important as feature sizes continue to shrink, with the possibility of faults affecting seemingly unrelated logic gates due to mask defects, shorts caused by metal migration or breakdown of insulation between layers, capacitive coupling, and so on. If a combinational circuit responds correctly to all possible combinations, then it has been satisfactorily tested for both the traditional stuck-at faults and for multiple faults as well. Unfortunately, for most circuits this is impractical (cf. Problem 4.1). Furthermore, some faults may still escape detection, such as those that change a combinational circuit into a sequential circuit. In addition, parametric faults that affect response time (i.e., delay faults) may escape detection if stimuli are applied at a rate slower than normal circuit operation.

In circuits where exhaustive testing is not feasible, alternatives exist. One alternative is to apply a random subset of the patterns to the circuit. Another alternative is to partition the circuit into combinational subcircuits. The smaller subcircuits can then be individually tested.[8] Additional tests can be added to test signal paths that were blocked from being tested by the partitioning circuits.

We look first at a cone of combinational logic that is to be tested using a subset of the pattern set. To understand this test strategy, consider a single detectable combinational fault in a cone of logic with m inputs. Since the fault is detectable, there is at least one vector that will detect it. Hence, if P_1 is the probability of detecting the fault with a single randomly selected vector, then

$$P_1 \geq 2^{-m}$$

To determine the probability of detecting the fault with n patterns, consider the binomial expansion

$$(a+b)^n = a^n + \binom{n}{1} a^{n-1} b^1 + \cdots + b^n$$

Let $a = 1 - 2^{-m}$ represent the probability of not detecting the fault.

Let $b = 2^{-m}$ represent the probability of detecting the fault.

Then, given n patterns, only the first term a^n in the expansion is totally free of the variable b. Hence, the probability P_n of detecting the fault with n patterns is

$$P_n \geq 1 - a^n = 1 - (1 - 2^{-m})^n$$

Note that this equation assumes true random sampling—that is, sampling with replacement. However, when using an LFSR of size equal to or greater than the number of circuit inputs, vectors do not repeat until all possible combinations have been generated. As a result, the above equation is somewhat pessimistic, predicting results that approach but never quite reach 100% coverage. Another factor that affects the probability of detection is the number of patterns that detect a fault. Consider a circuit comprised of an n-input AND gate. There are 2^n input combinations. Of these, only one input combination will detect a stuck-at-1 on the ith input. However, $2^n - 1$ patterns will detect a stuck-at-1 on the output of the AND gate. The stuck-at-1 on the output can be characterized as an "easy" to detect fault, in the sense that many patterns will detect it. The following equation takes into account the number of patterns that detect the fault:[9]

$$P_n = 1 - e^{-kL/N}$$

where k is detectability of the fault—that is, the number of patterns that detect the fault, L is the total number of vectors in the test, and N is 2^n, and n is the number of inputs to the circuit.

The expected coverage $E(C)$ is

$$E(C) = 1 - \frac{1}{n_f} \sum h_k e^{-kL/N}$$

In this equation, h_k is the number of faults with detectability k. In general, faults in real-world circuits tend to be detected by many vectors, resulting in large values of k. A drawback to this approach to computing effectiveness of BIST is the fact that the equation assumes a knowledge of the number of patterns that detect each fault. But that requires fault simulating the circuit without fault dropping, an expensive proposition. Nonetheless, this analysis is useful for demonstrating that fault coverage is, in general, quite good with just a few hundred pseudo-random vectors.[10]

9.4.2 Circuit Partitioning

The number of primary outputs in a circuit is another factor to be considered when attempting to determine fault coverage for pseudo-random vectors. A cone may be partially or completely subsumed by another cone, and the subsumed cone may

actually be exhaustively tested by the applied subset of vectors while the larger cone may receive fault coverage less than 100%. As a result the faults in the larger cone have different probabilities of detection, depending on whether they are in both cones or only the larger cone. An example of this is an ALU where the low-order bits may receive 100% fault coverage while high-order bits may have somewhat less than 100% coverage. In circuits where smaller cones are subsumed by larger cones (e.g., a functional block such as an ALU), there are frequently signals such as carries that lend themselves to partitioning. By partitioning the circuit at those signals, the partitioned blocks can be tested independent of one another to get improved coverage.

At first glance it may seem necessary to partition any circuit whose input count exceeds some threshold. But, partitioning may sometimes not be as critical as it at first appears; this is particularly true of data flow circuits.[11] Consider the 16-bit ALU in Figure 9.8. It is made up of 4-bit slices connected via ripple carries. The carry-out C_3 and the high-order output bit F_{15} would seem to be equally affected by all of the low-order bits. But the low-order bits only affect the high-order bits through the carry bits. For example, C_3 is clearly affected by C_2, but the probability that $A_{11} = 1$ and $B_{11} = 1$ is 0.25; hence the probability that C_2 is a 1 is $P_1(C_2) \geq 0.25$. Likewise, $P_0(C_2) \geq 0.25$. So, for this particular data flow function, C_3 is affected by the eight inputs A_{15-12}, B_{15-12} and a carry-in whose frequency of occurrence of 1s and 0s is probably around 50%. In this case, physically partitioning the circuit would probably not provide any benefit.

One of the barriers to getting good fault coverage with random patterns is the presence of gates with large fan-in and fan-out. To improve coverage, controllability and observability points can be added by inserting scan flip-flops in the logic, just as test points can be added to nonscan logic.[12] These flip-flops are used strictly for test purposes. Being in the scan path, they do not add to pin count. In Figure 9.9(a), the AND gate with large fan-in will have a low probability of generating a 1 at its output, adversely affecting observability of the OR gate; therefore a scan flip-flop is added to improve observability of the OR gate. The output of an AND gate with large fan-in can be controlled to a logic 1 by adding an OR gate, as shown in Figure 9.9(b), with one input driven by a scan flip-flop. During normal operation the flip-flop is at its noncontrolling value. These troublesome nets can be identified by means of a controllability/observability program such as SCOAP (cf. Section 8.3.1).

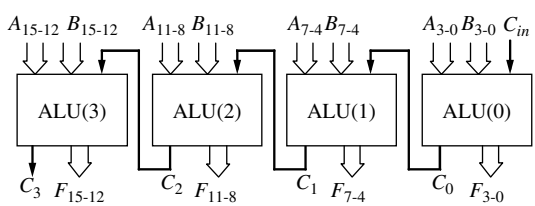


Figure 9.8 ALU with ripple carries.

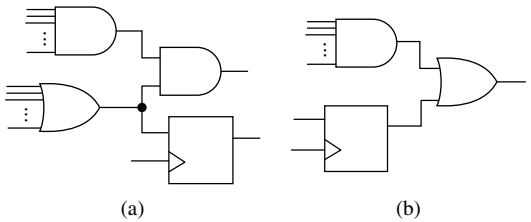


Figure 9.9 Enhancing random test.

9.4.3 Weighted Random Patterns

Another approach to testing random pattern-resistant faults makes use of *weighted random patterns (WRP)*. Sensitizing and propagating faults often require that some primary inputs have a disproportionate number of 1s or 0s. One approach developed for sequential circuits determines the frequency with which inputs are required to change. This is done by simulating the circuit and measuring switching activity at the internal nodes as signal changes occur on the individual primary inputs. Inputs that generate the highest amount of internal activity are deemed most important and are assigned higher weights than others that induce less internal activity.[13] Those with the highest weights are then required to switch more often.

A test circuit was designed to allocate signal changes based on the weights assigned during simulation. This hardware scheme is illustrated in Figure 9.10. An LFSR generates n-bit patterns. These patterns drive a 1 of 2^n selector or decoder. A subset j_k of the outputs from the selector drive *bit-changer k* which in turn drives input k of the IC, where $\sum_{k=1}^{m} j_k \leq 2^n$, and m is the number of inputs to the IC. The number j_k is proportional to the weight assigned to input k. The bit-changers are designed so that only one of them changes in response to a change on the selector outputs; hence only one primary input changes at the IC on any vector. When generating weights for the inputs, special consideration is given to reset and clock inputs to the circuit.

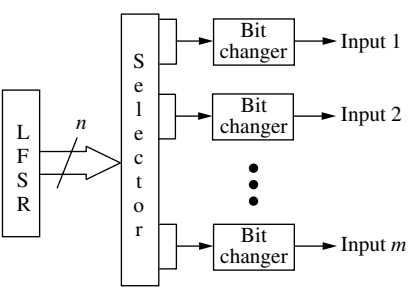


Figure 9.10 Weighted pattern generator.

The WRP is also useful for combinational circuits where BIST is employed. Consider, for example, a circuit made up of a single 12-input AND gate. It has 4096 possible input combinations. Of these, only one, the all-1s combination, will detect a stuck-at-0 at the output. To detect a stuck-at-1 on any input requires a 0 on that input and 1s on all of the remaining 11 inputs. If this circuit were being tested with an LFSR, it would take, on average, 2048 patterns before the all-1s combination would appear, enabling detection of a stuck-at-0 at the output. In general, this circuit needs a high percentage of 1s on its inputs in order to detect any of the faults. The OR gate is even more troublesome since an all-0s pattern is needed to test for a stuck-at-1 fault on the output, and the LFSR normally does not generate the all-0s pattern.

To employ WRPs on a combinational circuit, it is first necessary to determine how to bias each circuit input to a 1 or a 0. The calculation of WRP values is based on increasing the probability of occurrence of the nonblocking or noncontrolling value (NCV) at the inputs to a gate.[14] For the AND gate mentioned previously, it is desirable to increase the probability of applying 1s to each of its inputs. For an OR gate, the objective is to increase the probability of applying 0s to its inputs. The weighting algorithm must also improve the probability of propagating error signals through the gate.

The first step in computing biasing values is to determine the number of device inputs (NDI) controlling each gate in the circuit. This is the number of primary inputs and flip-flops contained in the cone of that gate. This value, denoted as NDI_g, is divided by NDI_i, the NDI for each input to that gate. That gives the ratio R_i of the NCV to the controlling value for each gate. This is illustrated in Figure 9.11, where the total number of inputs to gate D, NDI_D, is 9. NDI_A is 4; hence the ratio R_i of NDI_D to NDI_A is 9 to 4. Two additional numbers, W0 and W1, the 0 weight and the 1 weight, must be computed for each gate in the circuit. Initially, these two values are set to 1.

The algorithm for computing the weights at the inputs to the circuit proceeds as follows:

1. Determine the NDI_g for all logic gates in the circuit.
2. Assign numbers W0 and W1 to each gate; initially assign them both to 1.

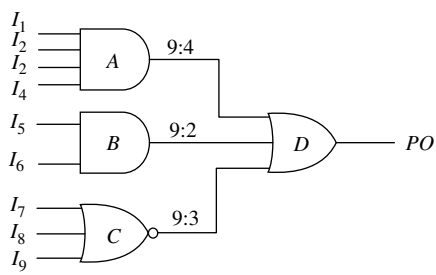


Figure 9.11 Calculating bias numbers.

TABLE 9.1 Weighting Formulas

Logic Function	$W0_i$	$W1_i$
AND	$W0_g$	$R_i \,.\, W1_g$
NAND	$W1_g$	$R_i \,.\, W0_g$
OR	$R_i \,.\, W0_g$	$W1_g$
NOR	$R_i \,.\, W1_g$	$W0_g$

3. Backtrace from each output. When backtracing from a gate g to an input gate i, adjust the weights W0 and W1 of gate i according to Table 9.1. When a gate occurs in two or more cones, the value of W0 or W1 is the larger of the existing value and the newly calculated value.
4. Determine the weighted value WV. It represents the logic value to which the input is to be biased. If W0 > W1, then WV = 0, else WV = 1.
5. Determine the weighting factor WF. It represents the amount of biasing toward the weighted value. If WV = 0, then WF = W0/W1, else WF = W1/W0.

Example Consider the circuit in Figure 9.11. Initially, all the gates are assigned weights W0 = W1 = 1. Then the backtrace begins. Table 9.2 tabulates the results. When backtracing from gate D to gate A, Table 9.1 states that if gate g is an OR gate, then $W0_i = (R_i \cdot W0_g)$ and $W1_i = W1_g$ for gate i. In this example, gate g is the OR gate labeled D and $W0_g = W1_g = 1$. Also, $R_i = 9/4$. Thus, $W0_i = 9/4$, or 2.25. In the next step of the backtrace, g refers to gate A, an AND gate, and i refers to primary inputs I_1 to I_4. Also, $R_i = 4/1 = 4$. The entry for the AND gate in Table 9.1 states that $W0_i = W0_g$ and $W1_i = (R_i \cdot WI_g)$. So the weights for I_1 to I_4 are $W0_i = 2.25$ and $W1_i = 4$. The remaining calculations are carried out in similar fashion.

From the results it is seen that inputs I_1 to I_4 must be biased to a 1 with a weighting factor WF = 4/2.25 = 1.77. Inputs I_5 and I_6 are biased to a 0 with WF = 4.5/2 = 2.25. Finally, inputs I_7 to I_9 have identical 0 and 1 weights, so biasing is not required for those inputs.

TABLE 9.2 Tabulating Weights

From (g)	To (i)	$W0_i$	$W1_i$
PO	gate D	1	1
gate D	gate A	2.25	1
gate A	$I_1 - I_4$	2.25	4
gate D	gate B	4.5	1
gate B	$I_5 - I_6$	4.5	2
gate D	gate C	3	1
gate C	$I_7 - I_9$	3	3

■■

The calculation of weights for a circuit of any significant size will invariably lead to fractions that are not realistic to implement. The weights should, therefore, be used as guidelines. For example, if a weight is calculated to be 3.823, it is sufficient to use an integer weighting factor of 4. The weighted inputs can be generated by selecting multiple bits from the LFSR and performing logic operations on them. An LFSR corresponding to a primitive polynomial will generate, for all practical purposes, an equal number of 1s and 0s (the all-0s combination is not generated). So, if a ratio 3:1 of 1s to 0s is desired, then an OR gate can be used to OR together two bits of the LFSR with the expectation that, on average, one out of every four vectors will have 0s in both positions. Similarly, for a ratio 3:1 of 0s to 1s the output of the OR can be inverted, or an AND gate can be used. ANDing/ORing three or four LFSR bits results in ratios of 7:1 and 15:1. More complex logic operations on the LFSR bits can provide other ratios.

When backtracing from two or more outputs, there is a possibility that an input may have to be biased so as to favor a logic 0 when backtracing from one output and it may be required to favor a logic 1 when backtracing from another output. How this situation is handled will ultimately depend on the method of test. If test patterns are being applied by a tester that is capable of biasing pseudo-random patterns, then it might be reasonable to use one set of weights for part of the test, then switch to an alternate set of weights. However, if the test environment is complete BIST, a compromise might require taking some average of the weights calculated during the backtraces. Another possible approach is to consider the number of inputs in each cone, giving preference to the cone with a larger number of inputs since the smaller cone may have a larger percentage of its complete set of input patterns applied.

Previously it had been mentioned that one approach to determining the weights on the inputs could be accomplished by switching individual inputs one at a time and measuring the internal activity in the circuit using a logic simulator. Another approach that has been proposed involves using ATPG and a fault simulator to initially achieve high-fault coverage.[15] Use these test vectors to determine the frequency of occurrence of 1s and 0s on the inputs. The frequency of occurrence helps to determine the weighting factors for the individual circuit inputs. It would seem odd to take this approach since one of the reasons for adopting BIST is to avoid the use of ATPG and fault simulation, but the approach does reduce or eliminate the reliance on a potentially expensive tester.

9.4.4 Aliasing

Up to this point the discussion has centered around how to improve fault coverage of BIST while minimizing the number of applied vectors. An intrinsic problem that has received considerable attention is a condition referred to as *aliasing*. If a fault is sensitized by applied stimuli, with the result that an error signal reaches an LFSR or MISR, the resulting signature generated by the error signal will map into one of 2^n possible signatures, where n is the number of stages in the LFSR or MISR. It is possible for the error signature to map into the same signature as the fault-free device. With 2^{16} signatures, the probability that the error signal generated by the fault will

be masked by aliasing is 1 out of 2^{16}, or about 0.0015%. If a functional register is being used to generate signatures and if it has a small number of stages, thus introducing an unacceptably high aliasing error, the functional register can be extended by adding additional stages that are used strictly for the purpose of generating a signature with more bit positions, in order to reduce the aliasing error.

9.4.5 Some BIST Results

The object of BIST is to apply sufficient patterns to obtain acceptable fault coverage, recognizing that a complete exhaustive test is impractical, and that there will be faults that escape detection. The data in Table 9.3 shows the improvement in fault simulation, as the number of random test vectors applied to two circuits increases from 100 to 10,000.[16]

For the sake of comparison, fault coverage obtained with an ATPG is also listed. The numbers of test patterns generated by the ATPG are not given, but another ATPG under similar conditions (i.e., combinational logic tested via scan path) generated 61 to 198 test vectors and obtained fault coverage ranging between 99.1% and 100% when applied to circuit partitions with gate counts ranging from 2900 to 9400 gates.[17]

9.5 SELF-TEST APPLICATIONS

This section contains examples illustrating some of the ways in which LFSRs have been used to advantage in self-test applications. The nature of the LFSR is such that it lends itself to many different configurations and can be applied to many diverse applications. Here we will see applications ranging from large circuits with a total commitment to BIST, to a small, 8-bit microprocessor that uses an ad hoc form of BIST.

9.5.1 Microprocessor-Based Signature Analysis

It must be pointed out here that BIST, using random patterns, is subject to constraints imposed by the design environment. For example, when testing off-the-shelf products such as microprocessors, characterized by a great deal of complex control logic, internal operations can be difficult to control if no mechanism is provided for that purpose. Once set in operation by an op-code, the logic may run for many clock

TABLE 9.3 Fault Coverage with Random Patterns

	Number of Gates	No. Random Patterns			Fault percentage with ATPG
		100	1000	10,000	
Chip1	926	86.1	94.1	96.3	96.6
Chip2	1103	75.2	92.3	95.9	97.1

cycles independent of external stimuli. Nevertheless, as illustrated in this section, it is possible to use BIST effectively to test and diagnose defects in systems using off-the-shelf components.

Hewlett-Packard used signature analysis to test microprocessor-based boards.[18] The test stimuli consisted of both exhaustive functional patterns and specific, fault-oriented test patterns. With either type of pattern, output responses are compressed into four-digit hexadecimal signatures. The signature generator compacts the response data generated during testing of the system.

The basic configuration is illustrated in Figure 9.12. It is a rather typical microprocessor configuration; a number of devices are joined together by address and data buses and controlled by the microprocessor. Included are two items not usually seen on such diagrams: a free-run control and a bus jumper. When in the test mode, the bus jumper isolates the microprocessor from all other devices on the bus. In response to a test signal or system reset, the free-run control forces an instruction such as an NOP (no operation) onto the microprocessor data input. This instruction performs no operation, it simply causes the program counter to increment through its address range.

Since no other instruction can reach the microprocessor inputs while the bus jumper is removed, it will continue to increment the program counter at each clock cycle and put the incremented address onto the address bus. The microprocessor might generate 64K addresses or more, depending on the number of address bits. To evaluate each bit in a stream of 64K bits, for each of 16 address lines, requires storing a million bits of data and comparing these individually with the response at the microprocessor address output. To avoid this data storage problem, each bit stream is compressed into a 16-bit signature. For 16 address lines, a total of 256 data bits must be stored.

The Hewlett-Packard implementation used the LFSR illustrated in Figure 9.2. Because testability features are designed into the product, the tests can be run at the product's native clock speed, while the LFSR monitors the data bus and accumulates a signature.

After the program counter has been verified, the ROM can be tested by running through its entire address space and generating a signature on each of its output pins.

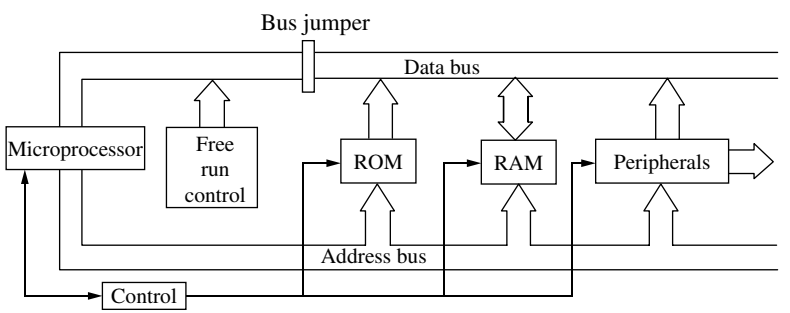


Figure 9.12 Microprocessor-based signature analysis.

The ROM, like the program counter, is run through its address space by putting the board in the free run mode and generating the NOP instruction. After the ROM has been checked, the bus jumper is replaced and a diagnostic program in ROM can be run to exercise the microprocessor and other remaining circuits on the board. Note that diagnostic tests can reside in the ROM that contains the operating system and other functional code, or that ROM can be removed and replaced by another ROM that contains only test sequences. When the microprocessor is in control, it can exercise the RAM using any of a number of standard memory tests. Test stimuli for the peripherals are device-specific and could in fact be developed using a pseudorandom generator.

The signature analyzer used to create signatures has several inputs, including START, STOP, CLOCK, and DATA. The DATA input is connected to a signal point that is to be monitored in the logic board being tested. The START and STOP signals define a window in time during which DATA input is to be sampled while the CLOCK determines when the sampling process occurs. All three of these signals are derived from the board under test and can be set to trigger on either the rising or falling edge of the signal. The START signal may come from a system reset signal or it may be obtained by decoding some combination on the address lines, or a special bit in the instruction ROM can be dedicated to providing the signal. The STOP signal that terminates the sampling process is likewise derived from a signal in the logic circuit being tested. The CLOCK is usually obtained from the system clock of the board being tested.

For a signature to be useful, it is necessary to know what signature is expected. Therefore, documentation must be provided listing the signatures expected at the IC pins being probed. The documentation may be a diagram of the circuit with the signatures imprinted adjacent to the circuit nodes, much like the oscilloscope waveforms found on television schematics, or it can be presented in tabular form, where the table contains a list of ICs and pin numbers with the signature expected at each signal pin for which a meaningful signature exists. This is illustrated for a hypothetical circuit in Table 9.4.

TABLE 9.4 Signature Table

IC	Pin	Signature	IC	Pin	Signature
U21	2	8UP3	U41	3	37A3
	3	713A		5	84U4
	4	01F6		6	F0P1
	7	69CH		8	1147
				9	8P7U
	9	77H1		11	684C
	11	10UP		15	H1C3
	14	1359			
	15	U11A			

During test the DATA probe of the signature analyzer is moved from node to node. At each node the test is rerun in its entirety and the signature registered by the signature analyzer is checked against the value listed in the table. This operation is analogous to the guided probe used on automatic test equipment (cf. Section 6.9.3). It traces through a circuit until a device is found that generates an incorrect output signature but which is driven by devices that all produce correct signatures on their outputs. Note that the letters comprising the signature are not the expected 0–9 and A–F. The numerical digits are retained but the letters A–F have been replaced by ACFHPU, in that order, for purposes of readability and compatibility with seven-segment displays.[19]

A motive for inserting stimulus generation within the circuits to be tested, and compaction of the output response, is to make field repair of logic boards possible. This in turn can help to reduce investment in inventory of logic boards. It has been estimated that a manufacturer of logic boards may have up to 5% of its assets tied up in replacement board kits and "floaters"—that is, boards in transit between customer sites and a repair depot. Worse still, repair centers report no problems found in up to 50% of some types of returned boards.[20] A good test, one that can be applied successfully to help diagnose and repair logic boards in the field, even if only part of the time, can significantly reduce inventory and minimize the drain on a company's resources.

The use of signature analysis does not obviate the need for sound design practices. Signature analysis is useful only if the bit streams at various nodes are repeatable. If even a single bit is susceptible to races, hazards, uninitialized flip-flops, or disturbances from asynchronous inputs such as interrupts, then false signatures will occur with the result that confidence in the signature diminishes or, worse still, correctly operating components are replaced. Needlessly replacing nonfaulted devices in a microprocessor environment can negate the advantages provided by signature analysis.

9.5.2 Self-Test Using MISR/Parallel SRSG (STUMPS)

STUMPS was the outcome of a research effort conducted at IBM Corp. in the early 1980s for the purpose of developing a methodology to test multichip logic modules.[21] The multichip logic module (MLM) is a carrier that holds many chips. The SRSG (shift register sequence generator) is their terminology for what is referred to here as a PRG.

Development of STUMPS was preceded by a study of several configurations to identify their advantages and disadvantages. The configuration depicted in Figure 9.13, referred to as a random test socket (RTS), was one of those studied. The PRG generates stimuli that are scanned into the MLM at the SRI (shift register input) pin. The bits are scanned out at the SRO (shift register output) and are clocked into a TRC to generate a signature. The scan elements are made up of LSSD SRLs (shift register latches). Primary inputs are also stimulated by a PRG, and primary outputs are sampled by a MISR. This activity is under control of a test controller that determines how many clock cycles are needed to load the internal scan chains. The

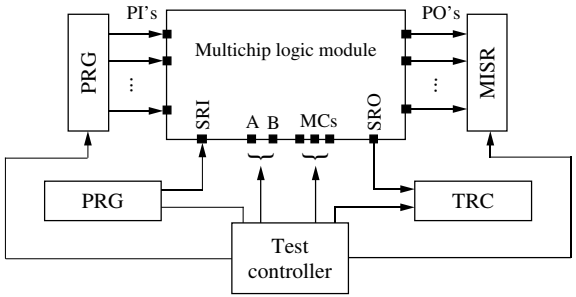


Figure 9.13 Random test socket.

test controller also controls the multichip clocks (MCs). When the test is done, the test controller compares the signatures in the MISR's to the expected signatures to determine if the correct response was obtained.

One drawback to the random test socket is the duration of the test. The assumptions are:

All of the SRLs are connected into a single scan path.

There would be about 10,000 SRLs in a typical scan chain.

The clock period is 50 ns.

About one million random vectors would be applied.

A new vector is loaded while the previous response is clocked into the MISR.

With these assumptions, the test time for an MLM is about 8 minutes, which was deemed excessive.

A second configuration, called simultaneous self-test (SST), converts every SRL into a self-test SRL, as shown in Figure 9.14(a). At each clock, data from the combinational logic is XOR'ed with data from a previous scan element, as shown in Figure 9.14(b). This was determined to produce reasonably random stimuli. Since every clock resulted in a new test, the application of test stimuli could be accomplished very quickly. The drawbacks to this approach were the requirement for a test mode I/O pin and the need for a special device, such as a test socket, to handle testing of the primary inputs and outputs.

A third configuration that was analyzed was STUMPS. The scan path in each chip is driven by an output of the PRG (recall from the discussion of LFSRs that a pseudo-random bit stream can be obtained from each SRL in the LFSR). The scan-out pin of each chip drives an input to the MISR. This is illustrated in Figure 9.15, where each chain from PRG to MISR corresponds to a one chip. The number of clocks applied to the circuit is determined by the longest scan length. The chips with shorter scan lengths will have extra bits clocked through them, but there is no penalty for that. The logic from the primary outputs of each chip drive the primary inputs to other chips on the MLM. Only the primary inputs and outputs of the MLM have to be dealt with individually from the rest of the test configuration.

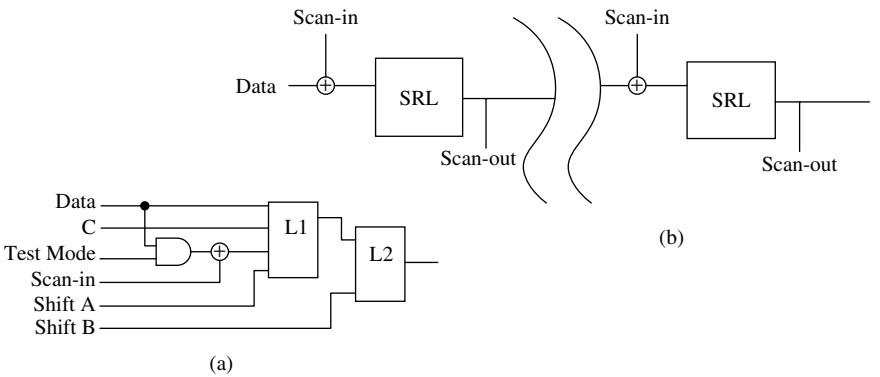


Figure 9.14 Simultaneous self-test.

Unlike RTS, which connects the scan paths of all the individual chips into one long scan path, scan paths for individual chips in STUMPS are directly connected to the PRG and the MISR, using the LSSD scan-in and scan-out pins, so loading stimuli and unloading response can be accomplished more quickly, although not as quickly as with SST. An advantage of STUMPS is the fact that, apart from the PRG and MISR, it is essentially an LSSD configuration. Since a commitment to LSSD has already been made and since STUMPS does not require any I/O pins in addition to those committed to LSSD, there is no additional I/O penalty for the use of STUMPS.

The PRG and MISR employed in STUMPS are contained in a separate test chip, and each MLM contains one or more test chips to control the test process. A MLM that contained 100 chips would require two test chips. Since the test chips are about the same size as the functional chips, they represented about a 2% overhead for STUMPS. The circuit in Figure 9.16 illustrates how the test chip generates the pseudo-random sequences and the signatures.

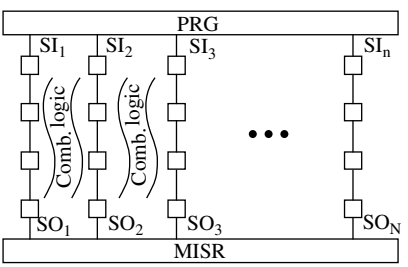


Figure 9.15 STUMPS architecture.

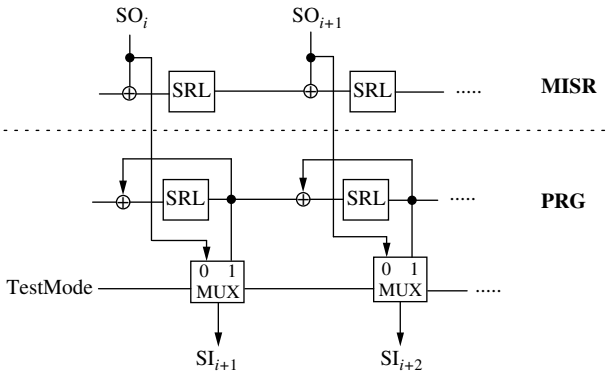


Figure 9.16 The MISR/PRG chip.

9.5.3 STUMPS in the ES/9000 System

STUMPS was used by IBM to test the ES/9000 mainframe.[22] A major advantage in the use of STUMPS was the ability to avoid creating the large test data files that would be needed if ATPG generated vectors and response were used to test the thermal conduction modules (TCM). A second advantage was simplification of TCM cooling during testing due to the absence of a probing requirement.

A typical STUMPS controller chip contained 64 channels. The fault coverage and the signatures generated by the circuits being tested were determined by simulation. Tests applied included a flush test, a scan test, an ABT test, and a logic test. The flush test (cf. Section 8.4.3) applies a logic 1 to both A and B clocks, causing all latches to be opened from the scan-in to the scan-out. Then a 1, followed by a 0, are applied to the scan chain input. This will reveal any gross errors in the scan chain that prevents propagation of signals to the scan output. The scan test clocks signals through the scan chain. The test is designed to apply all possible transitions at each latch.

In an ABT test the module is switched to self-test mode and the LFSR and MISR are loaded with initial values. Then all SRLs in the scan chains are loaded with known values while the MISR inputs are blocked. After the SRLs are loaded, the data are scanned into the MISRs. If the correct signature is found in the MISR, the STUMPS configuration is assumed to be working correctly. A correct signature provides confidence that the self-test configuration is working properly.

After the aforementioned three tests are applied and there is a high degree of confidence that the test circuits are working properly, the logic test mode is entered. STUMPS applies stimuli to the combinational logic on the module and creates a signature at the MISR. The tests are under control of a tester when testing individual modules. The tester applies stimuli to the primary inputs and generates signatures at the primary outputs. The input stimuli are generated by LFSRs in the tester, which

are shifted once per test. Response at primary outputs is captured by means of SISRs (single input signature registers) in the tester.

From the perspective of the engineers designing the individual chips, STUMPS did not require any change in their methodology beyond those changes required to accommodate LSSD. However, it did require changes to the Engineering Design System (EDS) used to generate test stimuli and compute response.[23] A compiled logic simulator was used to determine test coverage from the pseudo-random patterns. However, before simulation commences, design rule checking must be performed to ensure that X states do not find their way into the SRLs. If that happens, the entire MISR quickly becomes corrupted. Predictable and repeatable signatures was also a high priority.

For this particular development effort, the amount of CPU time required to generate a complete data file could range from 12 up to 59 hours. The data file for the TCM that required 59 hours to generate contained 152 megabytes and included test commands, signatures, and a logic model of the part. Fault coverage for the TCMs ranged from 94.5% up to 96.5%. The test application time ranged from 1.3 minutes to 6.2 minutes, with an average test time being 2.1 minutes.

Diagnosis was also incorporated into the test strategy. When an incorrect signature was obtained at the MISR, the test was repeated. However, when repeated, all chains but one would be blocked. Then the test would be rerun and the signature for each individual scan chain would be generated and compared to an expected signature for that chain. When the error had been isolated to one or more channels, the test would be repeated for the failing channels. However, this time it was done in bursts of 256 patterns in order to localize the failure to within 256 vectors of where it occured. RAM writes were inhibited during this process so the diagnostic process was essentially a combinational process. Further resolution down to eight patterns was performed, and then offline analysis was performed to further resolve the cause of the error signals. The PPSFP algorithm (Section 3.6.3) was used to support this process, simulating 256 patterns at a time.

The test time for a fault-free module was, on average, 2.1 minutes. Data collection on a faulty module extended the test time to 5 minutes. Diagnostic analysis, which included simulation time, averaged 11.7 minutes. Over 94% of faulty modules were repaired on the basis of automatic repair calls. Less than 6% of fails required manual analysis, and the resolution of the diagnostics averaged less than 1.5 chips per defect. This resulted, in part, from fault equivalence classes than spanned more than one chip.

9.5.4 STUMPS in the S/390 Microprocessor

Another product in IBM that made use of STUMPS was the S/390 microprocessor.[1] The S/390 is a single chip CMOS design. It incorporates pipelining and many other design features found in contemporary high-end microprocessors. In addition, it contains duplicate instruction and execution units that perform identical operations each cycle. Results from the two units are compared in order to achieve high data integrity. The S/390 includes many test features similar to those used in the ES/9000

system; hence in some respects its test strategy is an evolution of that used in the ES/9000. A major difference in approaches stems from the fact that ES/9000 was a bipolar design, with many chips on an MLM, whereas S/390 is a single-chip microprocessor, so diagnosing faulty chips was not an issue for S/390.

The number of tester channels needed to access the chip was reduced by placing a scannable memory element at each I/O, thus enabling I/Os to be controlled and observed by means of scan operations. Access to this boundary scan chain, as well as to most of the DFT and BIST circuitry, was achieved by means of a five wire interface similar to that used in the IEEE 1149.1 standard (cf. Section 8.6.2). An on-chip phase-locked loop (PLL) was used to multiply the tester frequency, so the tester could be run at a much slower clock speed. Because much of the logic dedicated to manufacturing test on the chips was also used for system initialization, recovery, and system failure analysis, it was estimated that the logic used exclusively for manufacturing test amounted to less than 1% of the overall chip area.

One of the motivating factors in the choice of BIST was the calculation that the cost of each full-speed tester used to test the S/390 could exceed $8 million. The choice of STUMPS permitted the use of a low-cost tester by reducing the complexity of interfacing to the tester. In addition, use of the PLL made it possible to use a much slower, hence less expensive, tester. BIST for memory test eliminated the need for special tester features to test the embedded memory. Another attraction of BIST is its applicability to system and field testing.

Because the S/390 is a single, self-contained chip, it was necessary to design test control logic to coexist on the chip with the functional logic. Control of the test functions is accomplished via a state machine within each chip, referred to as the self-test control macro (STCM). When in test mode, it controls the internal test mode signals as well as the test and system clocks. Facilities exist within the STCM that permit it to initiate an entire self-test sequence via modem. In addition to the BIST that tests the random combinational logic, known as LBIST (logic BIST), another BIST function is performed by ABIST (array BIST), which provides at-speed testing of the embedded arrays. An ABIST controller can be shared among several arrays. This both reduces the test overhead per array and permits reduced test times, since arrays can be tested in parallel. The STUMPS logic tests are supplemented by weighted random patterns (WRP) that are applied by the tester. Special tester hardware causes individual bits in scan-based random test patterns to be statistically weighted toward 1 or 0.

The incorporation of BIST in the S/390 not only proved useful for manufacturing and system test, but also for first silicon debug. One of the problems that was debugged using BIST was a noise problem that would allow LBIST to pass in a narrow voltage range. Outside that range the signatures were intermittent and nonrepeating, and they varied with voltage. A binary search was performed on the LBIST patterns using the pattern counter while running in the good voltage range. The good signatures would be captured and saved for comparison with the signatures generated outside the good voltage range. This was much quicker than resimulating, and it led to the discovery of the noisy patterns that had narrow good response voltage

windows. These could then be applied deterministically to narrow down the source of the noise.

LBIST was also able to help determine power supply noise problems. LBIST could be programmed to apply skewed or nonskewed load/unload sequences with or without system clocks. The feature was used to measure power supply noise at different levels of switching activity. LBIST was able to run in a continuous loop, so it was relatively easy to trace voltage and determine noise and power supply droop with different levels of switching activity. Some of these same features of LBIST were useful in isolating worst-case delay paths between scan chains.

9.5.5 The Macrolan Chip

The Macrolan (medium access controller) chip, a semicustom circuit, was designed for the Macrolan fiber-optic local area network. It consists of about 35,000 transistors, and it used BIST for its test strategy.[2] A cell library was provided as part of the design methodology, and the cells were able to be parameterized. A key part of the test strategy was a register paracell, which could be generated in a range of bit sizes. The register is about 50% larger than a scan flip-flop, and each bit contained two latches, permitting master/slave, edge-triggered, or two-phase, nonoverlapping clocking. All register elements are of this type, there are no free-standing latches or flip-flops. Two diagnostic control bits (DiC) from a diagnostic control unit permitted registers to be configured in four different modes:

User—the normal functional mode of the register
Diagnostic hold—contents of the register are fixed
Diagnostic shift—data are shifted serially
Test
- LFSR
- MISR
- Generate circular shifting patterns
- Hold a fixed pattern

When in test mode, selection of a particular test function is accomplished by means of two bits in the test register. These two bits, as well as initial seed values for generating tests, are scanned into the test register. Since the two control bits are scanned in, the test mode for each register in the chip can be individually selected. Thus, an individual scan chain can be serially shifted while others are held fixed.

The diagnostic control unit is illustrated in Figure 9.17. In addition to the clock (CLK), there are four input control signals and one output signal. Three other signals are available to handle error signals when the chip is used functionally. The chip select (CS) makes it possible to access a single chip within a system. Control (CON) is used to differentiate between commands and data. Transfer (TR) indicates that valid data are available and Loop-in is used to serially shift in commands or data. Loop-out is a single output signal.

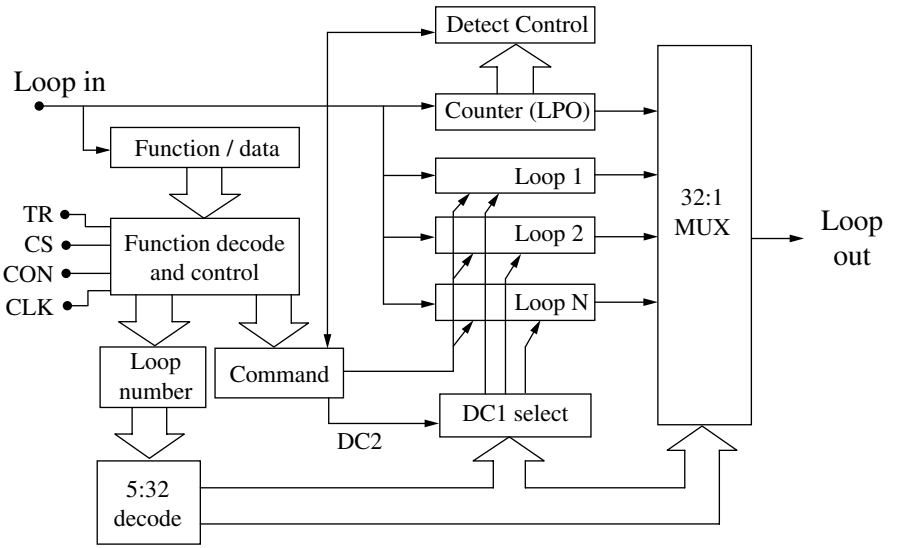


Figure 9.17 Macrolan diagnostic unit.

The diagnostic unit can control a system of up to 31 scan paths, each containing up to 128 bits. As previously mentioned, scan paths can be individually controlled using the two DiC bits. Scan path 0 is a 20-bit counter that is serially loaded by the diagnostic unit. It determines the number of clock cycles used for self-test; hence the system can apply a maximum of 2^{20} patterns. This limitation of 20 bits is imposed to minimize the simulation time required to compute signatures as well as to limit test time. The diagnostic unit can support chips using two or more clocks, but all registers must be driven from a master clock when testing the chip or accessing the scan paths.

The Macrolan chip makes use of a fence multiplexer to assist in the partitioning of the circuit. This circuit, illustrated in Figure 9.18, is controlled by a register external bit. During normal operation the register external bit is programmed to select input A, causing the fence to be logically transparent. When testing the chip, the fence plays a dual role. If input A is selected, the input to the fence can be compacted using the LFSR/MISR. When the external bit selects input B, the fence can be used in the generation of random patterns to test the logic being driven by the fence. Fences are also used to connect I/O pins to internal logic. This permits chips to be isolated from other circuitry and tested individually when mounted on a PCB.

Since the counter limits the number of tests to 2^{20}, a cone of combinational logic feeding an output cannot be tested exhaustively if it has more than 20 inputs. Since each output in a scan chain must satisfy that criteria with respect to the inputs to the

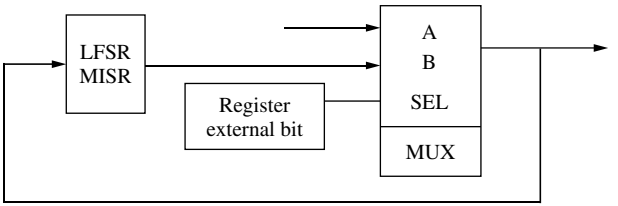


Figure 9.18 Fence multiplexer.

cone, and since logic cones, in general, are going to share inputs, a true exhaustive test for all the logic is virtually impossible to achieve. It is estimated that about 5% of the logic on the Macrolan chip is tested using exhaustive testing.

The BIST strategy employed by the design team made use of quasiexhaustive test. This test mode takes advantage of the observation that if $1 < N < 17$, where N is the number of inputs to a circuit, and if there are $M = 2^{N+3}$ random vectors (i.e., without replacement), then $P_M \geq 99.9\%$. Therefore, the LFSR can be cycled through a small subset of its patterns, with the result that there is no upper limit on the length of the LFSR, as there would be for an exhaustive test.

Another advantage to this mode of test is that two LFSRs can be used to generate patterns in parallel as long as their lengths are different. Consider two LFSRs of length A and B that generate maximal length sequences $S_A = 2^A - 1$ and $S_B = 2^B - 1$. The longest possible sequence generated by the two LFSRs running in parallel is $(2^A - 1) \times (2^B - 1)$, in which case their combined sequence will not repeat until both LFSRs return to their seed values simultaneously. The sequence length then will be the lowest common multiple of S_A and S_B, that is, $S_{A+B} = S_A \times S_B$. Put another way, the highest common factor (HCF) of S_A and S_B must be 1, which makes the sequence lengths of A and B coprime.

9.5.6 Partial BIST

Up to this point BIST has been discussed within the context of an all-or-nothing environment. But many test strategies employ BIST as one of several strategies to achieve thorough, yet economical test coverage. In particular, it is not uncommon to see designs where there is a sizable internal RAM that is tested using memory BIST or ROM that is tested by generating a signature on its contents while the random logic circuitry employs scan-based DFT. The PowerPC MPC750 is an example of a design that uses memory BIST (memory BIST will be discussed in Chapter 10). The MPC750 also employs functional patterns to test small arrays, clock modes, speed sorting, and other areas that were not fully tested by scan.[24]

The Pentium® Pro employed a BIST mode to achieve high toggle coverage for burn-in testing.[25] However, this feature was not intended to achieve high-fault coverage. Some LFSRs were used to support BIST testing of programmable logic arrays

(PLAs). Interestingly, their cost/benefit analysis led them to implement equivalent functionality in microcode for the larger PLAs. Signatures from the PLAs during BIST were acquired and read out using a proprietary Scanout mode under microcode control. In an earlier Intel paper describing the use of BIST for PLAs and microcode ROM (CROM), it was pointed out that the use of BIST during burn-in made it possible to detect a high percentage of early life failures.[26]

While there is growing interest in BIST, and it becomes easier to justify as circuits get larger and feature sizes get smaller, design teams have been able to justify it on the basis of cost/benefit analysis as far back as the early 1980s. The Motorola MC6804P2 is externally a small 8-bit microprocessor, but internally it is a serial architecture. It used BIST because it was determined to be cost effective as a test solution.[27] A 288-byte test program is stored in on-chip ROM; and an on-chip LFSR, using the CCITT-16 polynomial $x^{15} + x^{12} + x^5 + 1$, is updated at the end of each clock during the execution of the test program. A verify mode uses the same LFSR to test both customer and self-test ROM. The results are then compressed into a single 16-bit signature. The LFSR monitors the data bus so that during execution of the test program it is seldom necessary to perform compare and conditional branch instructions.

A flowchart for the MC6804P2 self-test is illustrated in Figure 9.19. The first step of the test checks basic ALU operations and writes results to a four-level stack. The ports and interrupt logic are then tested. The ports can be driven by a tester for worst-case test, or they can be tied together with a simple fixture for field test. After the test, the LFSR is read out and the 32-byte dynamic RAM is tested, and results are again read out. Then the RAM is filled with all-zeros, and those data are checked at the end of the test to confirm data retention. Again, after the timer test, the results are shifted out and a pass/fail determination is made. Finally, the data ROM test is used to test the data space ROM, the RAM that was previously cleared, the accumulator, and other miscellaneous logic.

The 288 bytes of test program are equivalent to about 1500 bus cycles. It was estimated that because of the serial nature of the microprocessor, each bus cycle was equivalent to about 24 clock cycles; hence the test would require about 36,000 test vectors. The customer ROM would add another 9000 vectors. Another factor impacting test size is the fact that the test program, if controlled by a tester, would need more compares, data reads, and so on, to replace the reads performed by the internal LFSR. Another motive for the BIST was its availability to customers.

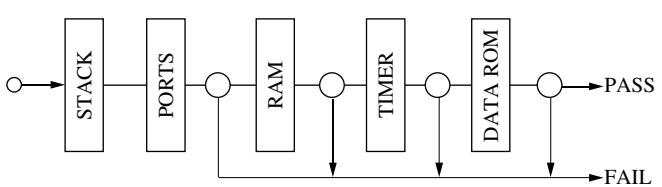


Figure 9.19 Self-test flowchart.

9.6 REMOTE TEST

Monitoring and testing electronic devices from a distant test station has been a fundamental capability for many years. However, it has tended to be quite expensive, and hence reserved for those applications where its cost could be justified. In former years it had been reserved for large, expensive mainframes and complex factory controllers. This mode of operation has recently migrated to more common devices, including the personal computer.

9.6.1 The Test Controller

In years gone by, the test controller was an indispensable part of the overall test strategy in many applications, including large mainframes and complex electronics systems for controlling avionics and factory operations, where a system might be comprised of several units, each comprised of many hundreds of thousands of logic gates. It might have different names and somewhat different assignments in different systems, but one thing the test controllers had in common was the responsibility to respond to error symptoms and help diagnose faults more quickly. Test controllers used some or all of the methods discussed in this and previous chapters, and they used some methods that will be discussed in subsequent sections. The general range of functions performed by the test controller include the following:

System startup
Communications with operator
System reconfiguration
Performance monitoring
System testing

Some of the earliest test controllers were used in support of mainframes. The typical test controller was a minicomputer or microprocessor. The rationale for this approach resulted from the fact that the mainframe was implemented using a technology such as emitter-coupled logic (ECL), which ran quite hot. As a result, it was much less reliable than the minicomputer or microprocessor that was used to apply test programs when the mainframe was powered up or when a problem occurred.

A typical system configuration is depicted in Figure 9.20. During system startup the test controller, or maintenance processor as it was sometimes called, was required to initialize the main processor, set or reset specific flip-flops and indicators, clear I/O channels of spurious interrupt requests, load the operating system, and set it into operation. Communication with the operator might result in operator requests to either conduct testing of the system or make some alterations to the standard configuration. A system reconfiguration might also be performed in response to detection of errors during operation. Detection of a faulty I/O channel, for example, might result in that channel being removed from operation and I/O activities for that channel being reassigned to another channel. Some or all of the reconfiguration was performed in conjunction with the main processor.

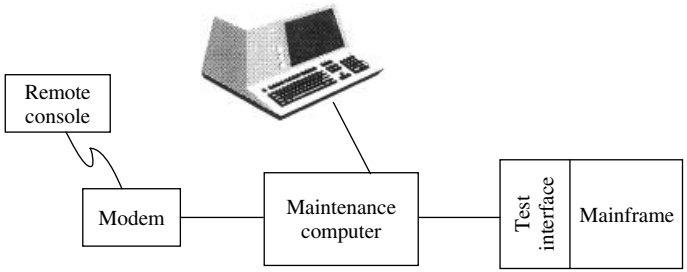


Figure 9.20 The maintenance processor.

Performance monitoring requires observing error indicators within a system during operation and responding appropriately. It is not uncommon for a maintenance processor to become aware of a problem before the computer operator realizes it. If an error signal is observed, an instruction retry may be in order. If the retry results in another error indication of the same nature, then a solid failure is indicated and a detailed test of some part of the system is necessary. The maintenance processor must determine what tests to select, and it must record the internal state of the system so that it can be restarted, whenever possible, from the point where the error was detected.

After applying tests, decisions must be made concerning the results of the tests. This may involve communicating with a field engineer either locally or, via remote link, at some distant repair depot. If tests do not result in location of a fault, but the error persists, then the field engineer may want to load registers and flip-flops in the system with specific test data via the maintenance processor, run through one or more cycles of the system clock, and read out the results for evaluation.

In conjunction with remote diagnosis, it is possible to maintain a database at a depot to assist the field engineer in those situations where the error persists but a fault cannot be located. The Remote Terminal Access Information Network (RETAIN) system is one such example.[28] It is a data base of fault symptoms that proved difficult to diagnose. It includes the capability for structuring a search argument for a particular product symptom to provide efficient and rapid data location. The data base is organized both on a product basis and on a symptom basis.

It should be noted that the maintenance processor must be verified to be working correctly. However, the computer chosen to serve as the maintenance processor was normally a mature product rather than a state-of-the-art device; it need not be fast, only reliable. Hence, it was generally orders of magnitude more reliable than the mainframe it was responsible for testing.

In microprogrammable systems implemented with writable control store, the maintenance processor can be given control over loading of control store. This can be preceded at system start-up time by first loading diagnostic software that operates out of control store. Diagnostics written at this level generally exercise greater control over internal hardware. Tests can run more quickly since they can be designed to

exercise functional units without making repeated instruction fetches to main memory. In addition, control at this level makes it possible to incorporate hardware test features such as BILBOs and similar BIST structures, and directly control them from fields in the microcode.

Maintenance processors can be given control over a number of resources, including power supplies and system clocks.[29] This permits power margining to stress logic components, useful as an aid in uncovering intermittents. Intermittents can also occasionally be isolated by shortening the clock period. With an increased system clock period, the system can operate with printed circuit boards on extender cards. Other reconfiguration capability includes the ability to disconnect cache and address translation units to permit operation in a degraded mode if errors are detected in those units.

The maintenance processor must be flexible enough to respond to a number of different situations, which suggests that it should be programmable. However, operating speed of the maintenance processor is usually not critical, hence microprocessor-based maintenance processors were used. One such system reported in the literature used a Z80 microprocessor.[30] The maintenance processor can trace the flow of activity through a CPU, which proves helpful in writing and debugging both diagnostic and functional routines in writable control store. Furthermore, the maintenance processor can reconfigure the system to operate in a degraded mode wherein an IPU (internal processor unit) that normally shares processing with the CPU can take over CPU duties if the CPU fails.

Another interesting feature of the maintenance processor is its ability to intentionally inject fault symptoms into the main processor memory or data paths to verify the operation of parity checkers and error detection and correction circuitry.[31] The logging of relevant data is an important aspect of the maintenance processor's tasks. Whenever indicators suggest the presence of an error during execution of an instruction, an instruction retry is a normal first response since the error may have been caused by an intermittent condition that may not occur during instruction retry.

Before an instruction retry, all data that can help to characterize the error must be captured and stored. This includes contents of registers and/or flip-flops in the unit that produced the error signal. Other parameters that may be relevant include temperature, line voltage, time, and date.[32] If intermittents become too frequent, it may be possible to correlate environmental conditions with frequency of occurrence of certain types of intermittent errors. If a given unit is prone to errors under certain stressful conditions, and if this is true in a large number of units in use at customer sites, the recorded history of the product may indicate an area where it may benefit from redesign.

The inclusion of internal busses in the mainframe to make internal operations visible is also supported.[33] An interesting addition to this architecture is the cyclic redundancy check (CRC) instruction, which enables both the operational programmer and the diagnostic programmer to generate signatures on data buffers or instruction streams.

The scan path can be integrated with the maintenance processor, as in the DPS88.[34] In this configuration the maintenance processor has access to test vectors

stored on disk. The tests may be applied comprehensively at system start-up or may be applied selectively in response to an error indication within some unit. The tests are applied to specific scan paths selectable from the maintenance processor. The scan path is first addressed and then the test vectors are scanned into the addressed serial path. Addressability is down to specific functional unit, board, and micropack (assembly on which 50 to 100 dice are mounted and soldered). The random pattern and signature features can be used in conjunction with the maintenance processor.[16]

9.6.2 The Desktop Management Interface

With the pace of technology permitting CMOS to overtake ECL technology, microprocessors with a clock period of 1.0 ns and less at the time of this writing are replacing mainframes of little more than a decade ago. The maintenance processor is not as common as it once was. However, the now ubiquitous personal computer (PC) has introduced a different set of problems. The mass production of millions of these PCs puts complex devices that are difficult to test and diagnose when they fail to work correctly into virtually every business and household. Furthermore, these PCs can be difficult to set up or alter if the owner wants to perform an upgrade. Clashes over software settings between application programs, or clashes over switch settings on the motherboard, can lead to significant frustration on the part of the owner of the PC.

A solution to this situation is the Desktop Management Interface (DMI). This is a specification defined by a consortium of vendors known as the Desktop Management Task Force (DMTF).[35] DMI 2.0 includes a remote management solution that makes it possible to access information across the internet by means of standard Remote Procedure Calls (RPC). The goal is to address cost of ownership problems. By developing standardized procedures for communicating between components of a system, it becomes possible to identify and report everything from simple operational problems, such as a device out of paper, to software problems such as conflicting interrupt settings, to hardware problems such as a CPU fan failure or an imminent hard disk head crash.

The general organization of the DMI is illustrated in Figure 9.21. The service layer collects information from the component interface, which in turn collects data from the hardware and software components. One of the components is an ASIC that collects data indicating excessive temperature, incorrect voltages, fan failures, and chassis intrusions. Information collected by the component interface is stored in a management information file (MIF) data base.

The management application gathers information from the MIF data base and the service layer via the management interface and reports the data by means of a graphical user interface (GUI). The management application can run on a remote console or on the client. System files for particular managed components can be updated when the product itself is being updated. Vendors of managed products provide the component interface—that is, test programs, data, and product attributes in MIF format. DMI requires a compatible BIOS that can communicate with the component interface and service provider. Some information, such as conflicting interrupt assignments or low memory or disk space, comes from the operating system.

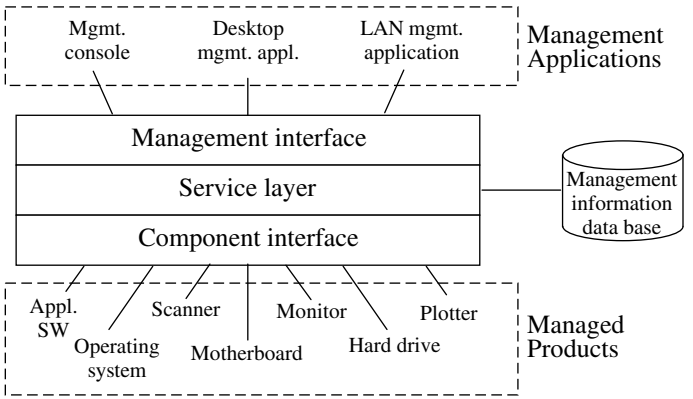


Figure 9.21 Desktop Management Interface (DMI).

Some of the information for DMI comes from operational modes that report such routine problems as paper jams, open cover, or low levels of toner or paper in a printer. Other information comes from more sophisticated analysis tools such as the hard drive reliability standard called Self-Monitoring, Analysis and Reporting Technology (SMART). This standard provides for on-drive sensing hardware for reporting drive status and software to collect and intrepret that data. The object is to measure physical degradation in the drive and alert the user to imminent failures. These measurements are recorded in the hard drive by sensor chips that potentially can measure up to 200 parameters such as head flying height, spin-up time, and so on. If a measurement falls outside of some predefined range, the drive issues an alarm that can be directed to the DMI which can display the measurement on its GUI.[36]

9.7 BLACK-BOX TESTING

This chapter began with a look at circuits designed to generate stimuli and accumulate response patterns, or signatures. These basic tools were then used, in conjunction with maintenance processors and scan methodologies, to test large mainframes. The solution was a global application of scan to all of the circuitry. We now turn our attention to the testing of circuits where, for various reasons, there is no visibility into the internal structure of the device or system. All testing is performed based on an understanding of the functionality of the device. Because of this lack of visibility, the methods described here are often referred to as *black-box testing*.

Testing of microprocessors and other complex logic devices can be aided by ordering and/or partitioning the functions within these devices. Ordering offers insight into the order in which functions should be tested. Furthermore, a good, robust ordering may suggest test strategies, since different partitions may lend

themselves to very different test methodologies. Where one partition may best be tested with BIST, another partition may be more effectively, or economically, tested using a conventional ATPG. A successful ordering of partitions may also be critical for those situations where detailed knowledge of the physical structure of the system is not available. In such cases, algorithmic test programs, such as those discussed in Chapter 7 for ALUs, counters, and so on, may be necessary. Within that context, it is necessary to develop a test program that is thorough while at the same time effective at diagnosing fault locations.

9.7.1 The Ordering Relation

A typical central processor unit (CPU) is illustrated in Figure 9.22. The figure is also typical of the amount of information provided by manufacturers of microprocessors. The information is usually provided for the benefit of the assembly language programmer. It displays a register stack, a control section, an ALU, a status register, a data bus, instruction register, and program counter. Information is provided on the architecture, including the instruction set, and a breakdown of the number of machine cycles required to execute the instructions.

Two or three decades ago, when the 8-bit microprocessor was dominant, it was not unusual to create a gate equivalent circuit and run ATPG. For contemporary, multi-million gate circuits, that is virtually impossible. An alternative is to resort to the use of block diagrams. In the method to be described here, a system is first partitioned into macroblocks, which are high-level functional entities such as CPUs, memory systems, interrupt processors, I/O devices, and control sections.[37] The macroblocks are then partitioned, to the extent possible, into smaller microblocks. Testing is organized at the microblock level, hence can be quite detailed, and can take into account the characteristics of the individual microcircuits. The objective is to obtain a comprehensive test for the microblock while using the macroblocks to route

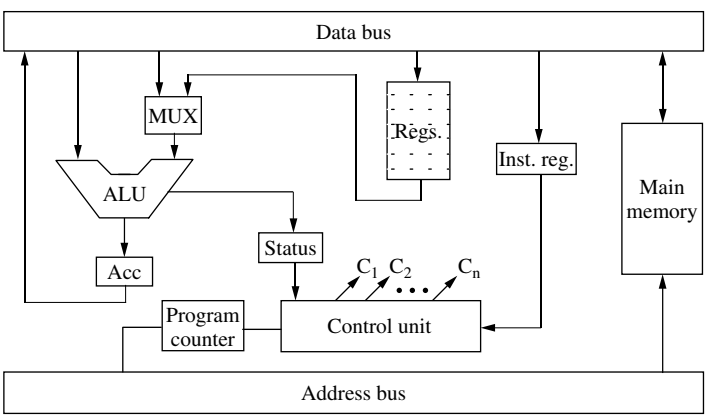


Figure 9.22 Typical central processor unit.

test information to observable outputs. When testing the microblocks in a given macroblock, all other macroblocks are assumed to be fault-free. Furthermore, the microblocks within a given macroblock are ordered such that a microblock is tested only through modules already tested.

Before discussing partitioning techniques for microblocks and macroblocks, we discuss the concept of hardcore. Hardcore circuits are those used to test a processor. *First-degree hardcore* is circuitry used exclusively for testing. It is verified independently of a processor's normal operational features and is then used to test the operational logic. Examples of first-degree hardcore include such things as a ROM dedicated to test which is loaded via a special access path not used by operational logic, a dedicated comparator for evaluating results, and watchdog timers that are used to verify that peripherals attached to the I/O ports respond within some specified time. A given device may or may not have first-degree hardcore. If it does, then the test strategy dictates that it be tested first. *Second-degree hardcore* is that part of the operational hardware used in conjunction with first-degree hardcore to perform test functions. Examples of this include the writable control store (WCS) used by test microprograms to exercise other operational units as well as the control circuitry and access paths of the WCS.

After first-degree hardcore has been verified, the second-degree hardcore is verified. Then the macroblocks are selected for testing. These are chosen such that a macroblock to be tested does not depend for its test on another macroblock that has not yet been tested. Individual microblocks within a chosen macroblock are selected for testing, again with the requirement that microblocks be tested only through other, previously tested microblocks. To achieve this, two ordering relations are defined. The controllability relation ρ_1 is defined by

$$A \cdot \rho_1 \cdot B \Leftrightarrow A \text{ can be controlled through } B$$

The observability relation ρ_2 is defined by

$$A \cdot \rho_2 \cdot B \Leftrightarrow A \text{ can be observed through } B$$

With these two relations, a priority partial order $\geq$ is defined such that

$$\text{If } B \cdot \rho_1 \cdot a \text{ and } B \cdot \rho_2 \cdot b, \text{ then } B \geq a \cdot b$$

In words, a test of B must follow the test of a AND b. In effect, if B is controlled through a and observed through b, then a and b must both be tested before B is tested. However, it may be that two devices C and D have the property that $C \geq D$ and $D \geq C$. In that case $A \equiv B$ and A and B are said to be *indistinguishable*. This would be the case, for example, if two devices were connected in series and could not possibly be tested individually. After a complete ordering has been established, the microblocks are partitioned into layers such that each microblock is tested only

through microblocks contained in previous layers. A microblock B is contained in a layer L_k if and only if

1. B follows at least one element of L_{k-1}
2. All elements smaller than B are contained in the union $\bigcup_{i=0}^{k-1} L_i$.

Layer L_0 is the hardcore; it is directly controllable and observable.

To assist in ordering microblocks, a tree is formed as illustrated in Figure 9.23. In that figure, the dot ($\cdot$) represents the AND operator and the plus (+) represents the OR operator. Therefore, $B \geq C \cdot D + E \cdot F$ states that the test of B must follow either the test of C AND D, OR it must follow the test of E AND F. In this graph, if an element occurs twice on the graph, with elements in between, then an indistinguishability block is defined that contains all elements joining the two occurrences of the element.

Example The ordering algorithm will be illustrated by means of the circuit in Figure 9.24. The various elements in that circuit are assigned numbers to identify them during the discussion that follows. We first identify the ρ_1 and ρ_2 relations:

Controlled by	Observed through
$1\ \rho_1\ 0$	$1\ \rho_2\ 0$
$2\ \rho_1\ 0$	$2\ \rho_2\ 4$
$3\ \rho_1\ 5$	$3\ \rho_2\ 4$
$4\ \rho_1\ 1 \cdot (2 + 3)$	$4\ \rho_2\ 0$
$5\ \rho_1\ 4$	$5\ \rho_2\ 3$

From these relations the following ordering relations can be derived:

$$1 \geq 0$$

$$2 \geq 4$$

$$3 \geq 4 \cdot 5$$

$$4 \geq 1 \cdot 2 + 1 \cdot 3$$

$$5 \geq 3 \cdot 4$$

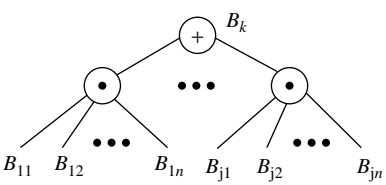


Figure 9.23 Ordering tree.

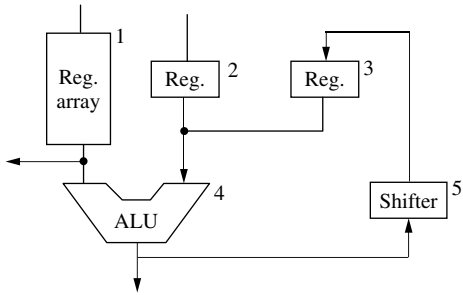


Figure 9.24 ALU circuit.

These relations in turn lead to the tree shown in Figure 9.25.

From the graph it can be seen that 1 does not follow any other microblock; therefore it is placed in layer L_1. It is also evident from the ordering relations that $2 \geq 4$ and $4 \geq 2$. That can also be see from the ordering tree. This implies an indistinguishability between 2 and 4. Therefore, a new block $b_1 = \{2,4\}$ is formed, and it replaces both 2 and 4. We get

$$b_1 \geq b_1$$
$$3 \geq b_1 \cdot 5$$
$$b_1 \geq b_1$$
$$b_1 \geq 1 \cdot b_1 + 1 \cdot 3$$
$$5 \geq 3 \cdot 4$$

Two reduction properties can be applied to the fourth relation, they are

$$c \geq c \cdot d + e \quad \text{implies} \quad c \geq d + e \tag{r1}$$
$$f \geq g + g \cdot h \quad \text{implies} \quad f \geq g \tag{r2}$$

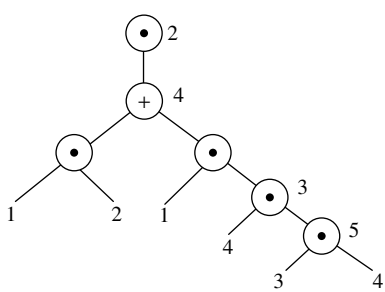


Figure 9.25 Ordering tree.

After applying these properties, the fourth relation becomes

$$b_1 \geq 1$$

The first and third relations ($b_1 \geq b_1$) are tautologies, so they can be eliminated and the ordering relations become

$$3 \geq b_1 \cdot 5$$
$$b_1 \geq 1$$
$$5 \geq b_1 \cdot 3$$

A microblock can be put in the present layer only if it appears solely on the right-hand side of the relation operator or if, whenever it appears on the left, the microblock(s) on the right are in lower level layers. In this case, microblock 1 is in a lower layer and in all other relations b_1 occurs only on the right. Therefore, it can be put in L_2. Setting $b_2 = \{3,5\}$ yields

$$b_2 \geq b_1$$
$$b_1 \geq 1$$
$$b_2 \geq b_1$$

The first and third relations can be reduced using relation (r1) to give

$$\text{b2} \geq b_1$$
$$b_1 \geq 1$$
$$b_2 \geq b_1$$

Then, b_2 can be placed in L_3.

All microblocks have now been placed into layers. Since the register array is in layer L_1, it should be tested first. This corresponds with the fact, seen in the diagram, that there is a separate path into and out of the register array. After it has been tested, it can be used to test the ALU and the register denoted as component 2, which were grouped together as indistinguishability block b_1. Finally, the shifter and the register denoted as component b_2 can be tested. ■■

Ordering a large number of microblocks within a macroblock can be tedious and time-consuming, and indistinguishability classes may become too complex. These complex classes may indicate areas in which additional hardware can be used to good advantage to break up loops and to improve controllability and observability.

9.7.2 The Microprocessor Matrix

The microprocessor generally absorbs a great deal of logic into a single IC. There may not be enough information to permit ordering the microblocks within a macroblock.

An alternate strategy[38] employs a matrix to relate the individual op-codes within the microprocessor to physical entities such as registers, ALUs, condition code registers, and I/O pins. A row of the matrix is assigned for each instruction. Several categories of columns are assigned; these include:

1. Data movement: register–register, register–memory, memory–immediate, and so on.
2. Operation type: AND, OR, COMPLEMENT, SHIFT, MOVE, ADD, SUBTRACT, MULTIPLY
3. I/O pins involved: data, address, control
4. Clock cycles involved: a column for each clock cycle
5. Condition codes affected: carry, overflow, sign, zero, parity, and so on.

If the *i*th instruction uses, affects, or is characterized by the property corresponding to column *j*, then there is a 1 in the matrix at the intersection of the *i*th row and *j*th column. A weight is assigned to each instruction by summing the number of 1s in the row corresponding to that instruction. Another matrix is created in which the rows represent functional units. Columns record such information as the number of instructions using the unit and any other physical information that is available, possibly including number of gate levels, number of feedback paths, and number of clocks in the unit. Note that the number of instructions that use a given unit may in many cases be a reasonable approximation to an ordering, in the sense in which an ordering was described in the previous subsection.

The test strategy requires that functional units with the lowest weight be tested first. Furthermore, the unit should be tested using, as often as possible, the instructions of minimum weight. The goal is to obtain a set of programs $\{P_i\}$ that test all of the functional units. Each program P_i has a weight that is equal to the sum of the weights of the individual instructions that make up the program. Because a given program may test two or more functional units, in the sense that two or more units are indistinguishable as defined in the previous subsection, a covering problem exists. The objective, therefore, given a set $\{P_i\}$ of programs that test all of the functional units, is to select the set of programs of minimum weight that cover (test) all functional units. A minimal weight test has the advantage that it can usually be applied more quickly, requires less memory, and reduces the likelihood that a fault will mask symptoms of another fault.

9.7.3 Graph Methods

The graph can also be used to show relationships between functional units of complex digital devices such as microprocessors.[39] This is illustrated in Figure 9.26, where paths are shown from some input source to the internal resources, and from internal resources to one or more output ports. Paths also exist between internal resources, and there are paths that loop back onto themselves. For example, in the hypothetical microprocessor of Figure 9.26, PC, which denotes program counter, has such a loop. A NOOP instruction (no-operation) simply involves incrementing

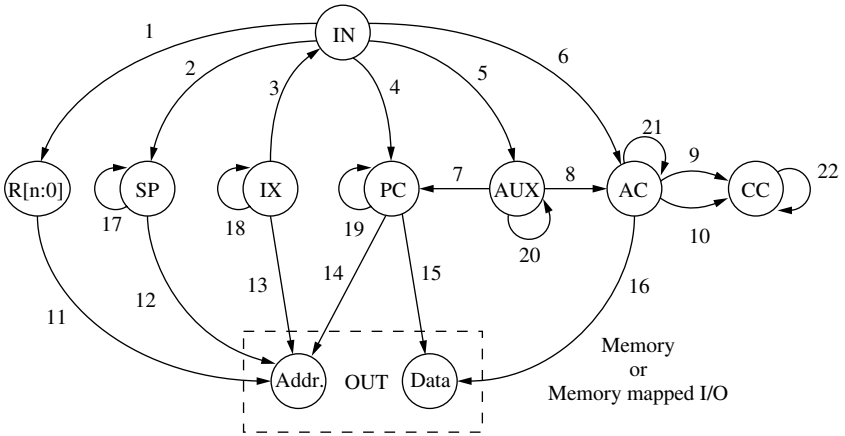


Figure 9.26 Graph model of hypothetical microprocessor.

the program counter to the next memory location. The accumulator (AC) can be incremented or decremented, or it can be used to receive the results of arithmetic and logic operations, and in the process the condition codes (CC) are updated. The individual arcs are numbered for convenience in referring to them.

If we denote an I/O port used for input as IN and denote an I/O port used for output as OUT, and if we assign graph nodes to IN and OUT, then a directed arc exists from IN to Reg. (from Reg. to OUT) if data transfer occurs, with or without transformation, from main memory or from an I/O port to register Reg. (from register Reg. to main memory or an I/O port). Further refinements are possible. Transformation devices such as counters and ALUs (arithmetic, logic unit) may be included in the graph. It must be recognized that these devices require more than simply passing data through them (cf. Section 7.8, Behavioral Fault Modeling).

9.8 FAULT TOLERANCE

If we distinguish between the logic machine, which is an abstract specification defining tasks to be performed and algorithms to perform them, and the host, which is the physical implementation of that abstract machine, then fault folerance can be defined as the architectural attribute of a digital system that permits the logic machine to continue performing its specified tasks when its physical host suffers various kinds of component failures.[40] We will look at some approaches to fault tolerance, but before looking at them, we distinguish between *active* fault tolerance and *passive* fault tolerance. Active fault tolerance is the ability to recover from error signals by repeating an operation, such as instruction retry, or rereading a data buffer or file, or requesting that a device retransmit a message. Passive fault tolerance is the ability to detect and correct errors without intervention by the host.

These are somewhat arbitrary distinctions since even in error detection and correction (EDAC) circuits, an error signal triggers logic activity in the hardware circuits of the host physical machine to correct the data, activity that would not have occurred if the error signal had not been detected. Perhaps a useful distinction is that active fault tolerance requires attention at the architectural level while passive fault tolerance contains errors before the symptoms are detected at the architectural level. In this text we will refer to active fault tolerance as *performance monitoring* since it more closely suggests the nature of the activities that take place.

The object of fault tolerance is to either prevent data contamination or to provide the ability to recover from the effects of data contamination. Applications range from data bases to industrial processes and transportation control. Consequences of faulty operation range from negligible to catastrophic. Hence the cost impact of fault-tolerant options employed may range from minor to significant. In some applications, such as space probes, it is rarely possible to repair faulty machines; hence cost for fault tolerance must be balanced against cost for failure of a critical part, which in turn must be equated with cost for failure of the entire mission.

9.8.1 Performance Monitoring

Performance monitoring involves the observation and evaluation of data during the course of normal operation. The monitoring may take advantage of information redundancy in the data or it may take advantage of structural characteristics of some particular functional units.

Parity Bit A parity bit is an example of monitoring information redundancy. It is claimed that in most digital systems, parity checking accounts for 70–80% of error detection coverage.[41] It can be applied to memory, control store, data and address buses, and magnetic tape storage. Parity bits can be appended to data transmitted between I/O peripherals and memory as well as to data transmitted via radio waves.

Signatured Instruction Stream A concept that requires an integrated software/hardware approach is the *signatured instruction stream*.[42] This approach, which can be applied to both microcode and program instructions, requires that a signature be generated on the stream of instructions coming out of memory or control store. Any branch or merge point in a set of instructions is accompanied by a signature generated by an assembler or compiler. The merge and branch points are illustrated in Figure 9.27. Each node represents an instruction. A merge node is any instruction that can be the successor to two or more instructions. In an assembler language program, most labeled instructions represent merge nodes. A branch node is an instruction, such as a conditional jump, that can have more than one successor.

The hardware computes a signature and then compares the computed signature against the signature provided by the assembler or compiler. If the signatures do not agree, there is an error in the instruction flow, either hardware or programming error, since self-modifying code is not permissible in this environment. When generating the signatures, it is necessary to reset the signature prior to a merge node since the

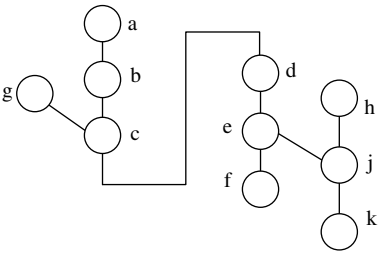


Figure 9.27 Graph representation of instruction stream.

value in the signature generator will normally be different for two paths converging at a common node. This is illustrated by instruction *j*, which could be executed in-line from instruction *h* or it could be reached by instruction *e* via a branch. Therefore, if *j* has a label, permitting it to be reached via a branch instruction, it is preceded by a special instruction that signals a check on the signature. Likewise, a branch instruction at *e* must cause the signature to be checked and reset.

The signature, being part of the instruction stream, must be designed in at the architectural level. Hardware and software must be designed and developed jointly. The signature is incorporated into the instruction stream by the assembler or compiler, which inserts an unconditional branch to location PC + 2 that causes the machine to skip the following two bytes during execution. A 16-bit embedded signature is inserted following the branch instruction. The special hardware recognizes the unconditional jump as being a signal that the next 16-bit word contains the signature. It can actually contain the inverse, so that the sum of the hardware calculated signature and the software calculated signature is zero. Then a nonzero value signals the presence of an error. Conditional jumps must also be considered. Since the instruction at node *e* may pass control to instruction *f* or instruction *j*, the signature generator must be resynchronized when going to instruction *f*.

A related scheme, called *branch address hashing*,[43] incorporates a signature into the branch address by performing a bit by bit exclusive-OR of the signature and the branch address. This permits a significant savings in program space and execution time. The branch address must, of course, be recovered before being used.

Diagnostic Programs In computers where priority scheduling and time sharing is employed, a maintenance program can reside in a part of memory and obtain a time slice of the CPU and other resources like any user program. When it receives control of the CPU, it executes special diagnostic procedures designed to test out as much of the machine as possible at the program level. If an error is detected during its performance, it can generate an interrupt to signal the operating system to load special diagnostic programs to further isolate the cause of the error. To avoid tying up resources during periods of peak computational demand, it can be a low-priority

task that runs only during off-peak time periods when resources are relatively inactive or during times when the program mix in memory is I/O intensive, permitting access to the CPU.

Test Data Injection It was previously pointed out that some maintenance processors are designed to inject test data into a circuit to specifically check parity checkers and other error dectection devices. Some architectures are particularly well suited to that operation. A single-instruction, multiple-data (SIMD) array processor, which performs identical calculations on multiple streams of incoming data, is one such example. During design of that hardware, time slots can be allocated for insertion of predetermined data samples into the data streams. The processing hardware then checks the received test data for correctness, knowing in advance what results to expect. This can verify most, if not all, hardware between the data capturing end and the processor.

9.8.2 Self-Checking Circuits

In some functions the output responses can be analyzed for correctness because some responses are simply not possible in a correctly operating circuit. If they occur, they indicate a malfunction. One such example is the 3-to-8 decoder. As designed, only a single output can be active for any 3-bit input. If two or more outputs are simultaneously active, there is an error in operation. If two OR gates are added to the outputs as shown in Figure 9.28, then the circuit becomes self-testing relative to the set of faults that either inhibit selection of an output line or cause two or more outputs to be simultaneously selected.[40] In general, a circuit is *self-testing* if any modeled fault eventually results in a detectable error.

If a circuit is designed so that during normal operation any modeled fault either does not affect the system's output or its presence is indicated no later than when the first erroneous output appears, then the circuit is said to be *fault-secure*. A majority logic decoder implemented with three AND gates and one OR gate, such that the output $M(a, b, c) = ab + bc + ac$, is fault-secure against opens on inputs since, during normal operation, all three input variables a, b, and c are identical. Therefore, a single open on a gate input will not affect the majority function output. The 3-to-8 decoder becomes fault-secure if the outputs are monitored so that an error signal occurs whenever more than one output is active. In fact, since it is both self-testing and fault-secure, it is said to be *totally self-checking*.[44]

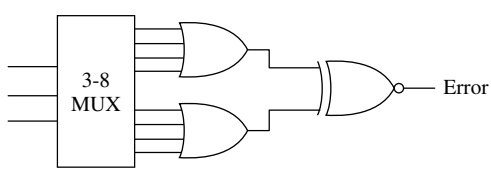


Figure 9.28 Self-testing decoder.

The multiplexer can be designed with self-testing features that take advantage of the function. The multiplexer must produce a logic 1(0) on its output if all data inputs are at logic 1(0), regardless of which input port was selected. In the 2-to-1 MUX shown in Figure 9.29, five gates are used to check for correct output from a three-gate circuit. However, only half of the input combinations can enable the error circuitry. For values of $n > 2$, the checking circuitry is more efficient in usage of components, since it still requires only five gates, but it is less efficient in percentage of input combinations that can enable the error detection circuitry.

State machines are candidates for self-checking.[45] The implementation style known as *one-hot encoding* assigns a flip-flop to each state in a state machine. Contrast the circuits in Figure 9.30 with the circuit in Figure 5.18 (defined by the state graph in Figure 5.16(a)). Figure 9.30(a) represents a canonical MUX implementation, with the state assignments listed below the circuit, while Figure 9.30(b) represents the equivalent one-hot encoding. Since one, and only one, flip-flop can have the value 1 in any clock period, the parity of the state flip-flops must always be 1. This fact can be exploited in two ways: First, a parity check of the state machine can detect errors immediately. Second, when fault simulating or performing ATPG on the state machine, there is instant observability through the parity check output. In fact, the parity checker can be connected to a parity tree, so that a single I/O can be used to monitor several state machines, as well as other logic.

9.8.3 Burst Error Correction

Error detection and correction (EDAC) codes are used with semiconductor memories in applications where errors cannot be tolerated. Such applications serve as examples of passive fault tolerance. If an error is detected, it is repaired "on-the-fly" by the EDAC circuitry; the processor is not aware that an error was detected and corrected. We will have more to say about this in Chapter 10. Error-correcting codes can also be used in an active fault tolerant role to correct burst errors in data transmitted from disk drives to main memory.[46] Disk packs have extremely thin coating of magnetic material. Errors occurring as a result of imperfections on a disk take the form of bursts. A type of code called Fire Codes, based on irreducible polynomials over GF(2), can correct single bursts in extremely long input streams.

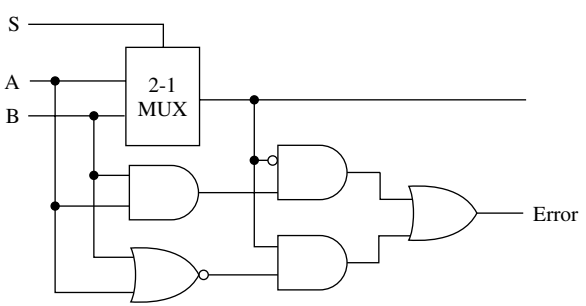


Figure 9.29 Multiplexer with self-test.

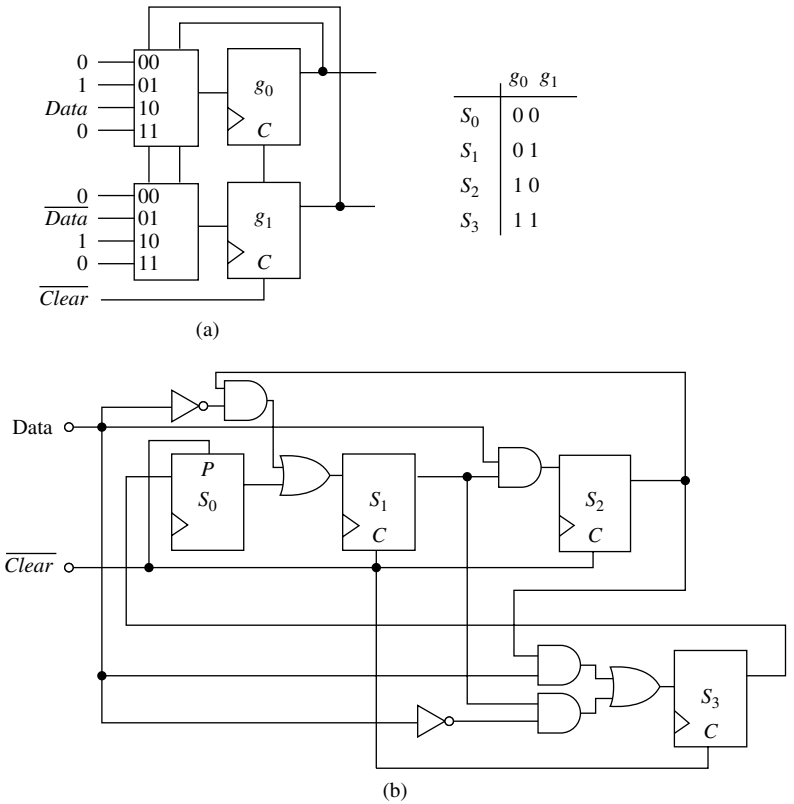


Figure 9.30 Mux (a) and one-hot encoding (b) implementations.

In what follows, $G(x)$ is defined to be a code generator polynomial and $M(x)$ is a message polynomial of degree $k - 1$. From the Euclidean division algorithm (a review of Section 9.3.1 might be helpful) we get

$$x^{n-k}M(x) = G(x) \cdot Q(x) + R(x)$$

where $M(x)$ is a message polynomial of degree $k-1$, $G(x)$ is the code generator polynomial, $Q(x)$ is the quotient, and $R(x)$ is the remainder. By virtue of modulo 2 arithmetic we have

$$G(x) \cdot Q(x) = x^{n-k}M(x) + R(x)$$

Therefore, $x^{n-k}M(x) + R(x)$ is a code vector for which the coefficients of $x^{n-k}M(x)$ of degree less than $n - k$ are zero and the remainder $R(x)$ has degree less than $n - k$. Therefore, in the codeword $x^{n-k}M(x) + R(x)$, $x^{n-k}M(x)$ is the original set of message bits and $R(x)$ is a set of check bits.

Recall from Section 9.2.1 that y was defined to be a root of $P(x)$ if $P(y) = 0$. The order of a polynomial was defined to be the smallest integer e such that $y^e = 1$, and a polynomial $P(x)$ was defined to be irreducible in GF(2) if there were no polynomials $P_1(x)$ and $P_2(x)$ with coefficients in GF(2) such that $P(x) = P_1(x) \cdot P_2(x)$.

Example Consider the residue class of polynomials modulo $G(x)$ over GF(2). If $a(x) = b(x) \cdot G(x) + c(x)$, then $a(x) \equiv c(x)$. Since $G(x) = a \cdot G(x) + 0$ for $a = 1$, x is a root of $G(x)$.

Let $G(x) = x^3 + x + 1$ over GF(2). The order of x is 7 since

$$x^7 = G(x) \cdot [x^4 + x^2 + x + 1] + 1 = 1 \bmod (G(x))$$

and no power of x of degree less than 7 has remainder equal to 1 when divided by $G(x)$. ■■

A Fire code is defined by its generator polynomial

$$G(x) = P(x) \cdot (x^c - 1)$$

where $P(x)$ is an irreducible polynomial over GF(2), of degree m, with roots of order $e = 2^m - 1$. It is also required that c not be divisible by e. The length n of the code is the least common multiple LCM(e,c) of c and e. We then have the following theorem.

Theorem 9.7 A vector that is the sum of a burst of length b or less and a burst of length d or less cannot be a code vector in a Fire code if

$$b + d - 1 \leq c$$

and m is at least as large as the smaller of b and d.

Proof We represent a burst of length b by a polynomial $x^i \cdot B(x)$, where degree$[B(x)] = b - 1$. We do likewise for $D(x)$. Then $F(x) = x^i \cdot B(x) - x^j \cdot D(x)$, where we assume, without loss of generality, that $i \leq j$. We use the Euclidean division algorithm, $j - i = cs + r$, $0 \leq r < c$, to get

$$F(x) = x^i[B(x) - x^r D(x)] - x^{i+r}[D(x)(x^{cs} - 1)]$$

We assume $F(x)$ is a codeword, so $j - 1 < n$, and $F(x)$ is divisible by $x^c - 1$. Therefore, the first term on the right is divisible by $x^c - 1$, so

$$B(x) - x^r D(x) = (x^c - 1) \cdot H(x)$$

where $H(x)$ is assumed to be nonzero. Then we get $r + d - 1 = c + h$, where h is the degree of $H(x)$. Using the inequality in the theorem, we get the result that $r \geq b + h$.

We also have that $b \geq 1$ and $h \geq 0$, so $r \geq b$ and $r > h$. $D(x)$ has a zero degree term; hence a term on the left has degree r and there is no term on the right with degree r because $h < r < c$. Hence we conclude that $r = 0$ and $H(x) = 0$, so $B(x) = D(x)$ and

$$F(x) = x^j D(x) \cdot (x^{cs} - 1)$$

Now, for $F(x)$ to be divisible by $P(x)$, it is necessary that $P(x)$ divide $B(x)$, $x^{cs} - 1$, or x^j. $P(x)$ cannot divide $B(x)$ since degree$[P(x)] = m \geq b >$ degree$[B(x)]$ and $P(x)$ is relatively prime to x; therefore $P(x)$ divides $F(x)$ if and only if $P(x)$ divides $x^{cs} - 1$. We have that e is the smallest number such that a root y of $P(x)$ satisfies $y^e = 1$. Therefore cs must be a multiple of e. But c is not divisible by e, and n is the least common multiple of c and e; therefore cs is a multiple of n, which is impossible.

The number of check bits in this code is $c + m$ and the number of information bits is $n - c - m$. The code can correct any burst of length $b \leq m$ and simultaneously detect any burst of length $d \leq b$, where $c \geq b + d - 1$.

The burst error processor is able to detect bursts because the factor $x^c - 1$ causes an evenly spaced interlacing of parity checks, so that the message symbols involved in any single-parity check bit are spaced c symbols apart. None of the c parity bits will be affected by more than a single error in any burst of length c or less. Hence, a single burst of c or less will be reproduced in the check bits.

Example Consider the AmZ8065 Burst Error Processor.[47] It has several different user-selectable polynomials, including

$g(x) = p(x) \cdot (x - 1) = (x^{11} + x^2 + 1) \cdot (x - 1)$
$e = 2^{11} - 1$
LCM$(21, 2^{11} - 1) = 42{,}987$
No. of check bits = 11 + 21 = 32
No. of information bits = 42,987 − 32 = 42,955
Correctable burst length = 11

The same chip has other polynomials that can correct single bursts of 11 bits in streams of up to 585,442 bits. The register length for correction is equal to the number of check bits; in this example, 32 flip-flops are required. The check symbols are generated by shifting a message polynomial $M(x)$ into a divider circuit such as the one shown in Figure 9.3, high-order bit first. After n shifts, k for the information symbols, and $n - k$ for the low-order zeros, the remainder is in the shift register. It is the modulo 2 inverse of the check symbols. The check symbols replace the low-order zeros to form the code vector. ■■

When a data stream is received, the nature of the received data

$$x^{n-k} M(x) + R(x) = G(x) \cdot Q(x)$$

implies that, after the complete data stream has been shifted through the decoding register, the parity bits $R(x)$ should be zero. If not zero, then an error has occurred. A correctible burst error $B(x)$ is of the form

$$E(x) = x^j B(x) = G(x) \cdot S(x) + R(x)$$

from which we get

$$\begin{aligned} x^i R(x) &= x^i x^j B(x) - x^i G(x) \cdot S(x) \\ &= (x^n - 1) \cdot B(x) - x^i G(x) \cdot S(x) + B(x) \end{aligned}$$

where we use $n = i + j$. Since $G(x)$ divides $x^n - 1$, $B(x)$ is the remainder after division of $x^i R(x)$ by $G(x)$.

This suggests the following decoding algorithm: Shift the received bits, including the parity bits, through a register identical to the decoder. If the register contains all zeros after the shift, there is no error. Otherwise, load the remainder $R(x)$ and shift until a burst $B(x)$ of length b or less occurs in the register. If such a burst does not occur, the error is uncorrectable. If a burst of length b or less occurs in the low-order b bits, and all zeros occur in the remaining bits, then the number of shifts that were applied to the remainder $R(x)$ to form the burst indicate where the burst occurred. At that point, the burst is added to the received message to correct it.

When data are required from a disk, the CPU normally initiates an I/O request and continues with other operations while an I/O processor supervises the read operation and, if required, the error correction. This illustrates the difficulty in classifying a method of fault tolerance as active or passive. The burst error correction may appear as passive fault tolerance to the CPU and as active fault tolerance to the I/O processor.

9.8.4 Triple Modular Redundancy

When designing a system, the cost of reliability must be balanced against the cost of system failure or occasional transients. On a video display, an occasional glitch may be barely perceptible. On a deep space probe, where maintenance is not possible, errors are intolerable. Enhanced reliability involves trade-offs: It may be a matter of using more reliable parts versus incorporating redundancy into a system; or, in a digital communications system, the choice may be the use of greater transmission power versus the use of error correcting codes. The cost for reliable parts, or a more powerful transmitter, tends to be a nonlinear function. To extend the mean-time-to-failure (MTTF) by 5% may cause the price of a component to double. In such a case, adding redundancy can produce a significant increase in availability of a system at less cost than would be had by employing more reliable parts.

The most obvious approach to fault tolerance is the use of triple modular redundancy (TMR). Using three identical computers, if R_m is the reliability of one machine, and all have the same reliability value, then

$$1 = [R_m + (1 - R_m)]^3 = (R_m)^3 + 3(R_m)^2 (1 - R_m) + 3R_m(1 - R_m)^2 + (1 - R_m)^3$$

The reliability of the system, then, assuming no errors in the voter circuits, is

$$R = \Pr(\text{no failures}) + \Pr(\text{one failure})$$

where $\Pr(x)$ denotes the probability of occurrence of x. From the previous equation we then get

$$R = (R_m)^3 + 3(R_m)^2(1 - R_m) = 3(R_m)^2 - 2(R_m)^3$$

Now, let the reliability of the circuit be a decaying exponential of the operating time:

$$R_m(t) = e^{-ft} = e^{-t/\text{MTTF}}$$

where f is the failure rate and MTTF, the reciprocal of f, is the mean time to failure. Then

$$\text{R(t)} = 3e^{-2t/\text{MTTF}} - 2e^{-3t/\text{MTTF}}$$

When $t >$ MTTF, $R < R_m$, hence triple modular redundancy actually degrades performance of the computer; that is, unreliable parts only make the situation worse. Since the computer is made up of functional units, each of which is more reliable than the entire computer, it suggests employing TMR at the level of functional units to obtain enhanced reliability.

The equations are generated on the assumption of perfect voter circuits. This is not unreasonable since the voter circuits are relatively simple circuits compared to the circuits whose outputs they are evaluating. It has been shown[48] that, since voter circuits are imperfect, maximum reliability can be achieved by using TMR on circuits at that functional level where reliability of the parts equals the reliability of the voters. However, this implies voting on circuits of approximately the same reliability as the voters themselves, which implies that the operational circuits being voted upon are of about the same size as the voters, which would lead not to a tripling of the logic but to a sixfold increase in the amount of logic needed to implement a machine.

The benefits of TMR can be enhanced by periodic maintenance. However, conventional testing and fault tolerance are at odds with one another. The goal of testing is to make a fault visible, while the goal of fault tolerance is to mask the effects of a fault. Therefore, when testing a unit it is necessary to disengage a module from its TMR environment or sample the outputs of the functional unit at some point prior to the voter circuits.

An example of fault tolerance with self-diagnosis capability is the Stratus/32 in which four processors run concurrently.[49] There are two processors on a board and two of each kind of board, including CPU, disk controller, and bus controller. If the pair of processors on a board disagree during operation, they remove themselves from operation and signal the system that they have failed. Maintenance software then runs tests on the board. If it passes, the maintenance routines assume the error

resulted from an intermittent and the board is restored to service. If the board fails again, it is removed from service until further, more extensive service can be provided. The other pair of processors takes over its tasks. The maintenance routines store all failure information in a log for inspection by a field engineer. A key requirement of the system is that failed boards be capable of being removed and replaced while the system is on line.

9.8.5 Software Implemented Fault Tolerance

Fault tolerance can be implemented at the software level. The SIFT (Software Implemented Fault Tolerance) system achieves fault tolerance by replication of tasks among processing units.[50] The primary method for detecting errors is through the detection of corrupt data. Interfaces between units are rigorously defined so as to deduce the effects of erroneous signals on a unit from a faulted unit. The level at which fault detection and reconfiguration are accomplished is a processor, memory module, or bus.

In the SIFT system, operation proceeds by execution of a set of tasks, each of which consists of a sequence of iterations. After executing an iteration, a processor places the results in its memory. A processor using the results will examine and compare the results from all processors performing that iteration. Discrepancies are recorded or analyzed by the executive system. An interesting concept is that of "loose coupling." Different processors executing the same iteration are not in lockstep synchronization, and may in fact be out of step by as much as 50 μs. Therefore, a transient in the system is not likely to affect all processors in the same way, thus increasing the likelihood that an error in the data caused by a transient will be discovered.

The number of processors performing an iteration can vary, depending on the importance of the task. A global executive determines relative importance of tasks. Spread of contaminated data is prevented by allowing a processor to write only into its own local memory. A processor reading data from the faulted unit will, when comparing that data, discover the error. Further protection from error is achieved by enabling a processor to acquire data not only from different processors, but via different buses as well.

In order to prevent incorrect control signals from producing wrong behavior in a nonfaulted unit, each unit is autonomous. In addition, the system has been designed to be immune to the failure of any single clock. The clock algorithm can be generalized such that a system can be made immune to failure of M out of N clocks when $N > 3M$.

9.9 SUMMARY

The ever-increasing complexity of digital circuits has spurred growing interest in the development of highly effective and economical test techniques. Design-for-test was explored at length in Chapter 8. Another approach that was explored is the

identification of invariant properties in functional units and general architectures, such as microprocessor-based architectures, in the hope that these invariants can lead to development of test methods applicable to general classes of architectures. Memory architectures stand out as one such candidate; more will be said about these in Chapter 10.

An alternate approach moves the tester into the logic, through the use of BIST circuits, thus testing numerous smaller functional units in parallel, while applying an exhaustive test, and testing at operational speed. This has the added benefit that it permits testing to be performed on site so that faults can be detected and repairs accomplished with a lower investment in expensive test equipment and spares inventory.

It was pointed out in Chapter 1 that two tests with X% coverage may not necessarily deliver the same acceptable quality level (AQL), as measured by customer returns.[51] A test composed of functional or design verification vectors may test most functions very thoroughly, but provide very low coverage for one or more functions. As a result, a defect in a function with low coverage may more likely escape detection. An advantage of BIST is the fact that, with pseudo-random patterns applied across the entire design, it is likely that all functions will get about the same coverage. So an X% total coverage is a more accurate indication of AQL for the overall circuit.

For large, complex systems, system availability is usually very high on the list of priorities. This requires both reliability, in order to prevent breakdowns, and maintainability to get the system back into operation quickly after it has failed. Maintainability, in turn, requires the ability to detect faults and diagnose their location quickly. Because availability requires confidence in the correctness of a system's operation, it is necessary that intermittent errors be minimized and traced to their origin. This has produced growing interest in fault-tolerant operation, including self-checking circuits, EDAC, and other performance monitoring techniques. The use of remote monitoring, in conjunction with centralized data bases such as the RETAIN data base previously discussed, is becoming an important adjunct to other methods. The data bases provide clues as to where design effort can be expended to improve reliability of the product.

Reliable delivery of system performance has been largely a hardware effort. The SIFT system, however, is one example of software fault tolerance. The signatured instruction stream is an example of hardware and software working together to provide reliable computing. While it is not within the scope of this book to address the subject of software correctness, nevertheless it was noted that the signatured instruction stream can occasionally detect software errors caused by unauthorized writing into program areas during program execution. Architectures can be developed specifically to enhance the detectability of software errors.[52] It has been shown that programming errors exist that are not detectable at compile time but which can be detected at execution time.

Much of what has been presented in this chapter has been a cursory overview. In practice, many additional factors must be considered when implementing a particular technique. The primary objective in this chapter was to survey the available

options and explain their salient features so the reader will know what choices are available and thus make an informed judgment as to which of them will fit his or her need. Successful application of any of the methods first requires establishing objectives, such as fast repair, reduction of spares inventory, less down time, fewer field returns, reduced test expense, or some combination of the above, as budget allows. Then, knowing the objectives, and knowing what techniques are available, it is possible to make an informed judgment as to what methods are best suited to one's particular problem and estimate the cost for that solution.

As BIST grows more prevalent, due to cost advantages, there is an emerging trend toward development of test modules in conjunction with functional modules and then to treat these as one integrated module.[53] For system on chip (SOC) it becomes possible to run tests on individual SOC blocks in parallel, with each functional block tested by its dedicated test circuits. All of the test circuits report back either to the tester or to a test controller. This can help to reduce the cost of test by getting the IC off the tester more quickly.

PROBLEMS

9.1 Given the two polynomials in Figure 9.31, compute the quotient and signature that result from the following bitstream (rightmost bit of string enters the LFSR first): 10000111011001001110101.

9.2 Given the LFSR and bitstream in Figure 9.32, if the bitstream is shifted into the LFSR, rightmost bit first, until all the bits are shifted in, how many single-bit errors in the bitstream will produce the same 4-bit signature as the error-free bitstream?

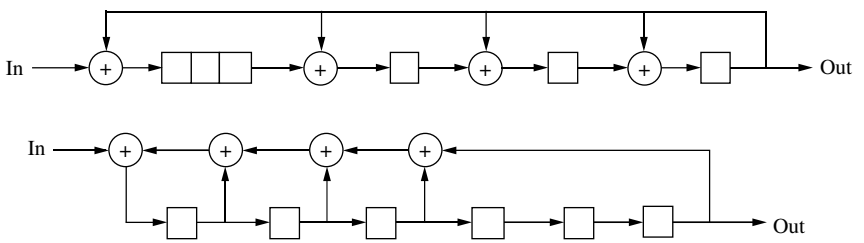


Figure 9.31 Two polynomials.

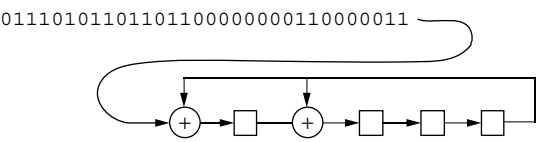


Figure 9.32 Polynomial and bitstream.

9.3 Given the following bitstreams, write a program to (a) count the number of transitions in each bitstream and (b) count the number of 1s in each bitstream.

```
01100110000101011010111000110111
11100010001111001111011011010000
00010100001011010101000111110111
10010010110000010011001000101000
11011011100000001111000111011111
10010000110001011011101000011010
10110011110010110110010000010001
00100110000001011000101001110110
01110001011111101010001011101101
```

9.4 It was stated Section 9.3.2 that pseudo-random bitstreams can be acquired from all stages of an LFSR in parallel. Can a single stage of a maximal length LFSR generate a sequence that begins to repeat sooner than other stages? If not, why?

9.5 Given the circuit in Figure 4.1, 51 faults are assigned: 37 stuck-at faults on the gate inputs, 12 stem faults on the six gates with multiple fanout, and two stem faults on the output of inverter *A*. A 5-bit counter applies all 32 combinations to the inputs. The 4-bit LFSR in Problem 9.3 is used to create a signature at the output. Approximately how many faults will escape detection?

9.6 Given a sum-of products circuit made up of three two-input AND gates driving a three-input OR gate, what is the probability of detection for a stuck-at-1 fault on an AND gate input when it is being tested by a six-stage LFSR? (i.e., I need a 01 combination on that AND gate, and every other AND gate must have at least one 0 on its inputs, so how many such combinations are there? Another way to look at this problem is to ask, How many times will that input be tested?)

9.7 Using the circuit in Figure 9.33, compute the biasing numbers for weighted random pattern generation.

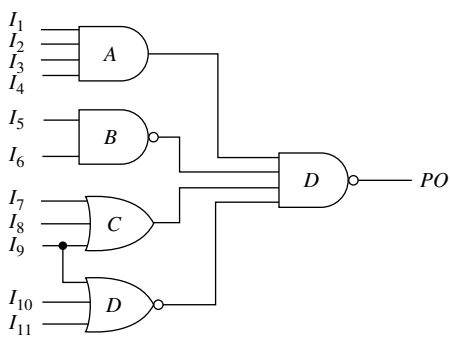


Figure 9.33 Biasing numbers.

9.8 The LFSR in Figure 9.34 corresponds to the polynomial $x^{12} + x^6 + x^4 + x + 1$. Write a program in C, Verilog or VHDL, to simulate it. Initialize stage 0 to 1, and all other stages to 0. Clock it until the initial value reappears. How many clock periods were required? Move the tap from position 6 to position 8 and simulate, then to position 9 and simulate. In each case, how many clocks are required to get back to the initial value?

9.9 Given the following two polynomials, which one is primitive?

$$x^6 + x^4 + x^2 + x + 1$$
$$x^6 + x^5 + x^2 + x + 1$$

9.10 Consider the 12-input OR gate discussed in Section 9.4.3. If you were using BIST and an LFSR to test the circuit, how would you detect a stuck-at-1 fault on the output of this circuit? If you were using an LFSR corresponding to the polynomial $x^{12} + x^6 + x^3 + x^2 + 1$ to test this circuit and if it was seeded with 1 in stage 0 and 0s in all other stages, what is the maximum achievable fault coverage for the 12-input OR gate? Would weighted random patterns help?

9.11 Prove that if a polynomial has an even number of terms, then it will detect all odd numbers of errors.

9.12 Prove that an LFSR corresponding to a polynomial of order n will detect all bursts of degree less than the polynomial.

9.13 When using checksum to detect errors, how many double errors go undetected when n words are being checksummed? How many triple errors?

9.14 Given an eight-input AND gate and a PRG that generates 128 patterns, using the equation $P_n = 1 - e^{-kL/N}$ in Section 9.4, what is the probable fault coverage for this circuit?

9.15 Given a circuit with eight inputs, and given a randomly selected fault, how many vectors must be applied to ensure that the probability of detection of that fault is greater than .99975? If the circuit has 16 inputs, how many vectors must be applied to get a probability of detection greater than .99975?

9.16 If the probability of detection of a single randomly selected fault is .99975, what is the probability that 100 such randomly selected faults will all be detected?

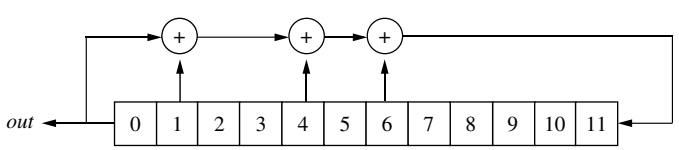


Figure 9.34 LFSR.

9.17 Design a bit-changer for the circuit in Figure 9.10 that causes a single bit to change on every clock.

9.18 Modify Figure 9.24 as follows: The outputs of registers 2 and 3 drive a multiplexer that has a control line. Label the multiplexer as unit 6. The shifter that drives register 3 now also feeds another register, labeled register 7, which is directly observable. Rework all calculations as a result of these changes.

9.19 For the ALU of Figure 9.22, draw a graph that shows the controllability/observability dependencies of the functional units.

9.20 Create a basic set of op-codes (e.g., ADD, OR, JUMP, etc.), for a hypothetical microprocessor such as that illustrated in Figure 9.26. For these op-codes, create a matrix of instructions versus functional units and status bits.

9.21 Determine the length of the Fire code generated by $g(x) = F(x) \cdot (x^{17} - 1)$.

9.22 Prove the reduction properties (r1) and (r2) in section 9.7.1.

REFERENCES

1. Huott, W. V. et al., Advanced Microprocessor Test Strategy and Methodology, *IBM J. Res. Dev.*, Vol. 41, No. 4/5, July/September 1997, pp. 611–627.
2. Illman, R., and S. Clarke, Built-in Self-Test of the Macrolan Chip, *IEEE Des. Test*, Vol. 7, No. 2, April 1990, pp. 29–40.
3. www.dmtf.org/spec/dmis.html
4. Meggett, J. E., Error Correcting Codes and Their Implementation for Data Transmission Systems, *IRE Trans. Inf. Theory*, Vol. IT-7, October 1961, pp. 234–244.
5. Smith, J. E., Measures of the Effectiveness of Fault Signature Analysis, *IEEE Trans. Comput.*, Vol. C-29, No. 6, June 1980, pp. 510–514.
6. Nebus, J. F., Parallel Data Compression for Fault Tolerance, *Comput. Des.*, April 5, 1983, pp. 127–134.
7. Konemann, B. et al., Built-in Logic Block Observation Techniques, *Proc. IEEE Int. Test Conf.*, 1979, pp. 37–41.
8. McCluskey, E. J., Verification Testing—A Pseudoexhaustive Test Technique, *IEEE Trans. Comput.*, Vol. C-33, No. 6, June 1984, pp. 265–272.
9. McCluskey, E. J. et al., Probability Models for Pseudorandom Test Sequences, *Proc. IEEE Int. Test Conf.*, 1987, pp. 471–479.
10. Wagner, K. D., and E. J. McCluskey, Pseudorandom Testing, *IEEE Trans. Comput.*, Vol. C-36, No. 3, March 1987, pp. 332–343.
11. Illman, Richard J., Self-Tested Data Flow Logic: A New Approach, *IEEE Des. Test*, April 1985, Vol. 2, No. 2. pp. 50–58.
12. Eichelberger, E. B., and E. Lindbloom, Random-Pattern Coverage Enhancement and Diagnosis for LSSD Logic Self-Test, *IBM J. Res. Dev.*, Vol. 27, No. 3, May 1983, pp. 265–272.
13. Schnurmann, H. D. et al., The Weighted Random Test-Pattern Generator, *IEEE Trans. Comput.*, Vol. c-24, No. 7, July 1975, pp. 695–700.

14. Waicukauski, J. A. et al., A Method for Generating Weighted Random Test Patterns, *IBM J. Res. Dev.*, Vol. 33, No. 2, March 1989, pp. 149–161.
15. Siavoshi, F., WTPGA: A Novel Weighted Test-Pattern Generation Approach for VLSI Built-In Self Test, *Proc. IEEE Int. Test Conf.*, 1988, pp. 256–262.
16. Eichelberger, E. B., and E. Lindbloom, Random-Pattern Coverage Enhancement and Diagnosis for LSSD Logic Self-Test, *IBM J. Res. Dev.*, Vol. 27, No. 3, May 1983, pp. 265–272.
17. Laroche, G., D. Bohlman, and L. Bashaw, Test Results of Honeywell Test Generator, *Proc. Phoenix Conf. Comput. Commun.*, May 1982.
18. Hewlett-Packard Corp., *A Designer's Guide to Signature Analysis*, Application Note 222, April, 1977.
19. Nadig, H. J., Testing a Microprocessor Product Using a Signature Analysis, *Proc. Cherry Hill Test Conf.*, 1978, pp. 159–169.
20. White, Ed, Signature Analysis-Enhancing the Serviceability of Microprocessor-Based Industrial Products, *Proc. IECI*, March 1978, pp. 68–76.
21. Bardell, P. H., and W. H. McAnney, Self-Testing of Multichip Logic Modules, *Proc. IEEE Int. Test. Conf.*, 1982, pp. 200–204.
22. Bardell, Paul H., and Michael J. Lapointe, Production Experience with Built-in Self-test In the IBM ES/9000 System, *Proc. IEEE Int. Test Conf.*, 1991, pp. 28–36.
23. Keller, B. L., and T. J. Snethen, Built-in Self-test Support in the IBM Engineering Design System, *IBM J. Res. Dev.*, Vol. 34, March/May 1990, pp. 406–415.
24. Pyron, C. et al., Next Generation PowerPC™ Microprocessor Test Strategy Improvements, *Proc. IEEE Int. Test Conf.*, 1997, pp. 414–423.
25. Carbine, A. et al., Pentium® Pro Processor Design for Test and Debug, *Proc. IEEE Int. Test Conf.*, 1997, pp. 294–303.
26. Gelsinger, P., Design and Test of the 80386, *IEEE Des. Test*, June 1987, pp. 42–50.
27. Kuban, J. R., and W. C. Bruce, Self-Testing the Motorola MC6804P2, *IEEE Des. Test*, May 1984, pp. 33–41.
28. Hsiao, M. Y. et al., Reliability, Availability, and Serviceability of IBM Computer Systems: A Quarter Century of Progress, *IBM J. Res. Dev.*, Vol. 25, No. 5, September 1981, pp. 453–465.
29. Wallach, S., and C. Holland, 32-Bit Minicomputer Achieves Full 16-Bit Compatability, *Comput. Des.*, January 1981, pp. 111–120.
30. Hawk, R. L., A Supermini for Supermaxi Tasks, *Comput. Des.*, September 1983, pp. 121–126.
31. Boone, L. et al., Availability, Reliability and Maintainability Aspects of the Sperry Univac 1100/60, *Proc. 10th Fault Tolerant Computing Symp.*, October 1980, pp. 3–8.
32. Frechette, T. J., and F. Tanner, Support Processor Analyzer Errors Caught by Latches, *Electronics*, November 8, 1979, pp. 116–118.
33. Swarz, R. S., Reliability and Maintainability Enhancements for the VAX-11/780, *Proc. 8th Fault Tolerant Computing Symp.*, June 1978, pp. 24–28.
34. Miller, H. W., Design for Test Via Standardized Design and Display Techniques, *Electron. Test*, Vol. 6, No. 10, October 1983, pp. 108–116.
35. http://www.dmtf.org

36. Nicolaisen, Nancy, I'm Failing and I Can't Boot Up!, *Byte Magazine*, October, 1997, pp. 112NA1–112NA6.
37. Robach, C., G. Saucier, and J. Lebrun, Processor Testability and Design Consequences, *IEEE Trans. Comput*., June 1976, Vol. C-25, No. 6, pp. 645–652.
38. Srini, V. P., Fault Diagnosis of Microprocessor Systems, *Computer*, Vol. 10, No. 1, January 1977, pp. 60–65.
39. Thatte, S. M., and J. A. Abraham, Test Generation for Microprocessors, *IEEE Trans. Comput*., Vol. C-29, No. 6, June 1980, pp. 429–441.
40. Avizienis, A., Fault-Tolerance: The Survival Attribute of Digital Systems, *Proc. IEEE*, Vol. 66, No. 10, October 1978, pp. 1109–1125.
41. Bossen, D. C., and M. Y Hsiao, Model for Transient and Permanent Error-Detection and Fault-Isolation Coverage, *IBM J. Res. Dev.*, Vol. 26, No. 1, January 1982, pp. 67–77.
42. Sridhar, T., and S. M. Thatte, Concurrent Checking of Program Flow in VLSI Processors, *Proc. IEEE Int. Test Conf.*, 1982, pp. 191–199.
43. Shen, J. P., and M. A. Schuette, On-Line Self-Monitoring Using Signatured Instruction Streams, *Proc. IEEE Int. Test Conf.*, 1983 pp. 275–282.
44. Smith, J. E., A Theory of Totally Self-Checking System Design, *IEEE Trans. Comput*., Vol. C-32, No. 9, September 1983, pp. 831–844.
45. Miczo, A., A Self-Test Hardwired Control Section, *IEEE Trans. Comput*., Vol. C-32, No. 7, July 1983, pp. 695–696.
46. Lignos, Demetrios, Error Detection and Correction in Mass Storage Equipment, *Comput. Des*., October 1972, pp. 71–75.
47. AmZ8065 Product Specification, Advanced Micro Devices,Sunnyvale, CA, 94086.
48. Lyons, R. E., and W. Vanderkulk, The Use of Triple-Modular Redundancy to Improve Computer Reliability, *IBM J*., April 1962, pp. 200–209.
49. Hendrie, G., A Hardware Solution to Part Failures Totally Insulates Programs, *Electronics*, January 29, 1983, pp. 103–105.
50. Wensly, J. H. et al., SIFT: Design and Analysis of a Fault Tolerant Computer for Aircraft Control, *Proc. IEEE*, Vol. 66, No. 10, October 1978, pp. 1240–1255.
51. Maxwell, Peter C., Reductions in Quality Caused by Uneven Fault Coverage of Different Areas of an Integrated Circuit, *IEEE Trans. CAD*, Vol. 14, No. 5, May 1995, pp. 603–607.
52. Myers, G. J., *Advances in Computer Architecture*, Chapter 13, John Wiley & Sons, New York, 1978.
53. Zorian, Yervant, Embedded Test Complicates SoC Realm, http://www.eedesign.com/story/0EG20001222s0049.

CHAPTER 10

Memory Test

10.1 INTRODUCTION

Memory is pervasive in digital products. Consider, for example, the personal computer (PC). It has main memory, video memory, translation ROMs, shadow ROMs, scratchpad memory, hard disk, floppy disk, CDROM, and various other kinds of storage distributed throughout. In addition, the die that contains the microprocessor may also contain one or more levels of cache.

A typical PC is depicted in the block diagram of Figure 10.1. It is basically a memory hierarchy connected by several buses and adapters and controlled by a CPU. The purpose for much of the hierarchy is to combine two or more storage systems with divergent capacities, speeds, and costs such that the combined system has almost the speed of the smaller, faster, more expensive memory at almost the cost, speed, and storage capacity of the larger, slower, less expensive memory. Clearly, not all storage devices are part of this hierarchy. The CDROM may be used to deliver programs and/or data to an end user, and video memory is dedicated to the display console. The central processing unit (CPU) accesses many of these auxiliary memory devices through a peripheral component interconnect (PCI) bus, which regulates the flow of data through the system.

Unlike the random logic that has been considered up to this point, memory storage devices are characterized by a high degree of regularity. For example, a semiconductor memory is organized as an array of cells, while storage on a hard drive is organized into cylinders. This regularity of semiconductor memories permits much greater packing of transistors on die. For example, in the PowerPC MPC750, memory accounts for 85% of the transistors but only 44% of the die area.[1] In the Alpha 21164, 80% of the 9.6 million transistors are used for three on-chip caches, but the remaining 20% of the transistors occupy a majority of the physical die area.[2] The various storage devices in Figure 10.1 employ different kinds of circuits for storing and retrieving data, and different kinds of media for retaining data, hence they have unique failure mechanisms, requiring different test strategies. These memories may also employ varying levels of redundancy to detect and/or correct errors during operation.

Digital Logic Testing and Simulation, *Second Edition*, by Alexander Miczo
ISBN 0-471-43995-9 Copyright © 2003 John Wiley & Sons, Inc.

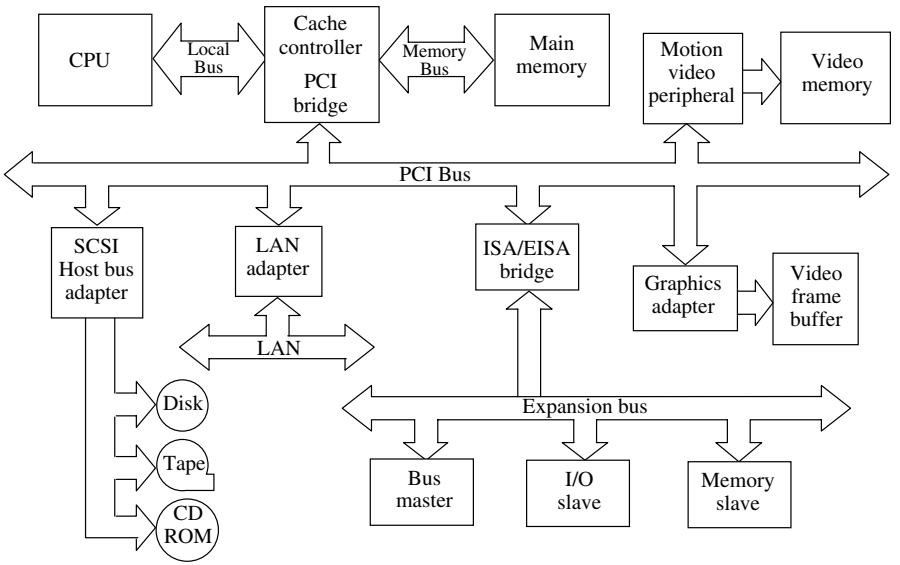


Figure 10.1 Memory distribution in a typical PC architecture.

10.2 SEMICONDUCTOR MEMORY ORGANIZATION

Because semiconductor memories are characterized by a high degree of regularity, it is easy to devise algorithms to test them. However, because of the growing capacity of memories, many of the tests will run for unacceptably long periods of time. A significant problem then, when testing memories, is to identify the kinds of faults that are most likely to occur and determine the most efficient tests for those faults.

Semiconductor memories can be characterized according to the following properties:

Serial or random access
Volatile or nonvolatile
Static or dynamic
Destructive or nondestructive readout.

Serial access memories are those in which data are accessed in a fixed, predetermined sequence. Magnetic tape units are an example of serial access. To read a record it is necessary to read the entire tape up to the point where the desired data exists. By way of contrast, a *random* access memory (RAM) permits reading of data at any specific location without first reading other data. When performing a read of a FIFO (first-in, first-out) memory, the first location stored is the first to be read out. These memories act as buffers when transferring data between functional units with different data rates. A stack in a computer, often used to save data and return addresses, is an example of a LIFO (last-in, first-out) memory. The last data pushed onto the stack is the first data to become available when the stack contents are popped from the stack.

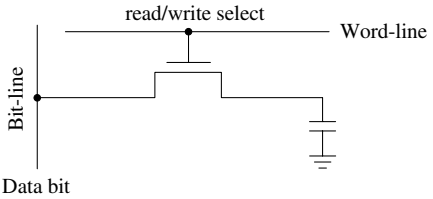


Figure 10.2 Dynamic memory cell.

Memories can be categorized according to whether or not they can retain information when power is removed. A *nonvolatile* memory can retain information when power is removed. Examples of nonvolatile memories include magnetic cores, magnetic tapes, disks, MROMs, EPROMS, EEPROMS, and flash memories. *Volatile* memory devices lose information when power is removed.

Volatile memories can be further broken down into static and dynamic memories. A *static* memory retains information as long as power is applied, while a *dynamic* memory can lose information even when power is continuously applied. Static RAMs (SRAMs) are flip-flops that, with their two stable states, can remain in a given state indefinitely, without need for refresh, as long as power is applied; that is, they are static but volatile. The dynamic RAM (DRAM), illustrated in Figure 10.2, is an example of a dynamic memory. The cell is chosen if decoding the memory address causes its word-line to be selected. It is basically a capacitor that can either be discharged onto the bit-line or that can be recharged from the bit-line. Since it is a capacitor, the charge can leak away over time. The memory system must employ refresh circuitry that periodically reads the cells and writes back a suitably amplified version of the signal.

If the contents of a memory device are destroyed by a read operation, it is classified as a *destructive readout* (DRO); otherwise it is a *nondestructive readout* (NDRO) device. DRAMs must be refreshed when their contents are read out, since a read causes the capacitor to discharge.

Programmable read-only memories (PROMs) are slightly more complicated to characterize. They are static and nonvolatile. Mask programmable ROMs and fuse programmable ROMs are programmed once and thereafter can only be read. EPROMs (erasable PROMs) can be erased by means of ultraviolet light, which involves physically removing them from the system in which they are installed. For all practical purposes, they are programmed only once because it is quite inconvenient to erase and reprogram them, unless they are being used to emulate a new design for the purposes of debugging that design.

EEPROMs (electrically erasable PROMs) can be reprogrammed after being installed in a system, but their response time is slower than DRAMs or SRAMs; hence they are confined to applications where nonvolatility is required. Flash memories are structurally almost identical to EPROMs, but they can be reprogrammed in a system and are more dense than EEPROMs. However, EEPROMs can be programmed a bit at a time, whereas flash memories are erased a block at a time before being reprogrammed. The Venn diagram in Figure 10.3 illustrates this distribution of properties among the various kinds of semiconductor memories.[3]

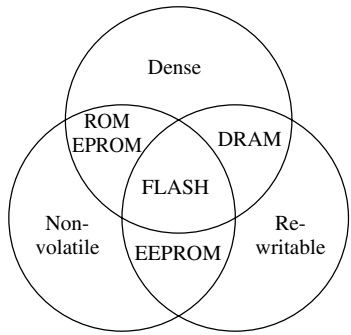


Figure 10.3 Semiconductor memory properties.

Semiconductor memories usually employ an organization called 2-D. In this organization a $2^m \times 1$ memory with m address lines is organized into a matrix with 2^N rows and 2^M columns ($N + M = m$). The address lines are split into two groups such that N lines go to a row decoder and M lines go to a column decoder. This is illustrated in Figure 10.4. The row decoder selects 2^N memory cells, and the column decoder selects one of those to be read out of or written into memory. This idealized organization is the subject of numerous modifications whose purpose is to permit faster operation and/or faster test. One of the more significant changes is the division of the memory array into several smaller arrays. This reduces loading on the bit lines. As we shall see, it also permits multiple cells to be tested simultaneously.

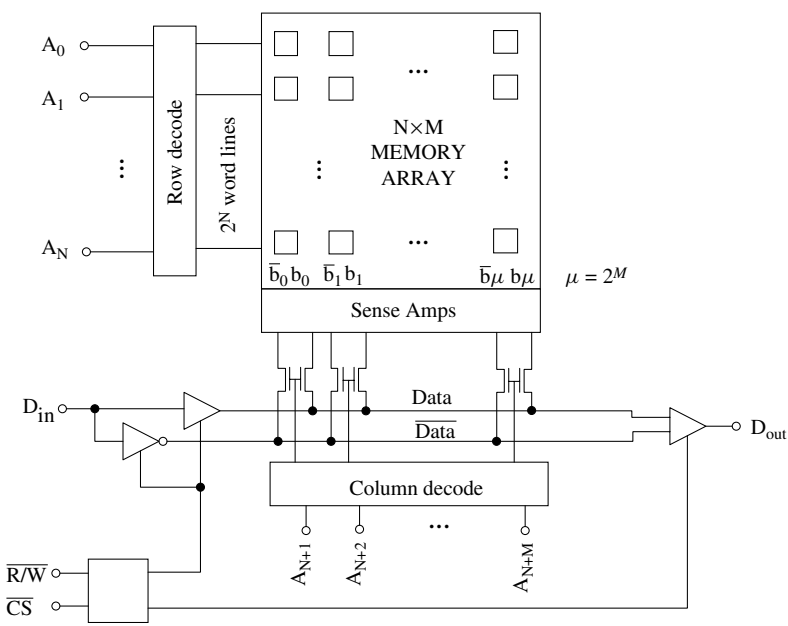


Figure 10.4 A semiconductor memory organization.

10.3 MEMORY TEST PATTERNS

In this section some classical, or legacy, memory test algorithms will be examined. Memory test algorithms fall into two categories: functional and dynamic. A *functional test* targets defects within a memory cell, as well as failures that occur when cell contents are altered by a read or write to another cell. A *dynamic test* attempts to find access time failures. The *All 1s* or *All 0s* tests are examples of functional tests. These tests write 1s or 0s into all memory cells in order to detect individual cell defects including shorts and opens. However, these tests are not effective at finding other failure types.

A memory test pattern that tests for address nonuniqueness and other functional faults in memories, as well as some dynamic faults, is the *GALPAT* (GALloping PATtern), sometimes referred to as a ping-pong pattern. This pattern accesses each address repeatedly using, at some point, every other cell as a previous address. It starts by writing a background of zeroes into all memory cells. Then the first cell becomes the test cell. It is complemented and read alternately with every other cell in memory. Each succeeding cell then becomes the test cell in turn and the entire read process is repeated. All data are complemented and the entire test is repeated. If each read and compare is counted as one operation, then GALPAT has an execution time proportional to $4N^2$, where N is the number of cells. It is effective for finding cell opens, shorts, address uniqueness faults, sense amplifier interaction, and access time problems.

The following Verilog code illustrates the operation of the GALPAT test. First, a RAM module of size "memdepth" $\times$ 1 bit is described. The RAM model contains code used to insert a stuck-at fault at memory location 27. The RAM model is followed by a testbench that executes the GALPAT test. The line of code that instantiates the RAM passes parameters into the RAM from the testbench in order to override the RAM size.

```
module ram(addr, datai, datao, wen, oen);
parameter log2_memdepth = 8, memdepth = 256;
input [log2_memdepth–1:0] addr;
input datai, wen, oen;
output datao;
reg ramcore[memdepth – 1:0];
reg datao;
always @(oen or wen or addr)
  begin
    if (!oen && wen)    datao = ramcore[addr];
    else if (oen)       datao = 1'bz;
    else                datao = 1'bx;
  end
always @(negedge wen)
  begin
```

```
    if(addr == 27)  // inject a fault at location 27
      ramcore[addr] = 1'b1;
    else
      ramcore[addr] = datai;
  end
endmodule
module testbench;
parameter log2_memdepth = 6;
parameter memdepth = 64;
reg  [log2_memdepth-1:0] addr;
reg  datain, wen, oen, memval;
wire dataout;
integer e, i, j;
ram #(log2_memdepth,memdepth)
       U1(addr,datain,dataout,wen, oen);
always
  begin
    for(e = 0; e <= 1; e = e+1)
    begin
      for(i = 0; i < memdepth; i = i+1)
        write(e,i);  // write background of e,  e ∈ {0,1}
      for(i = 0; i < memdepth; i = i+1)
      begin
        write(!e,i);
        for(j = 0; j < memdepth; j = j+1)
          if(j != i)
          begin //check all mem. loc. except loc. i
            read(memval, j);
            if(memval != e)
              $display("Mem. Error at loc. %d\n",j);
          end // for j
        read(memval,i);  // loc. i should not change
        if(memval != !e)
          $display("Mem. Error at loc. %d\n", j);
        write(e,i);  // restore value at loc. i
      end // for i
    end   // for e
    $finish;
  end    // always
  task write;      // write to memory
```

```
  input data, adval;
  integer adval;
    begin
      datain = data;
      addr   = adval;
      #1 wen = 0; #1 wen = 1;
    end
  endtask
  task read;   // read from memory
  output data;
  input adval;
  integer adval;
    begin
      addr = adval;
      #1 oen = 0;
      #0.5 data = dataout;
      #0.5 oen = 1;
    end
  endtask
endmodule
```

Walking Pattern is similar to the GALPAT except that the test cell is read once and then all other cells are read. To create a Walking Pattern from the GALPAT program, omit the second read operation in the testbench. The Walking Pattern has an execution time proportional to $2N^2$. It checks memory for cell opens and shorts and address uniqueness.

March, like most of the algorithms, begins by writing a background of zeroes. Then it reads the data at the first location and writes a 1 to that address. It continues this read/write procedure sequentially with each address in memory. When the end of memory is reached, each cell is read and changed back to zero in reverse order. The test is then repeated using complemented data. Execution time is of order N. It can find cell opens, shorts, address uniqueness, and some cell interactions.

Galloping Diagonal is similar to GALPAT in that a 1 is moved through memory. However, it is moved diagonally, checking both row and column decoders simultaneously. It is of order $4N^{3/2}$. Row and column GALPATs of order $4N^{3/2}$ also exist.

Sliding Diagonal (see Figure 10.5) writes a complete diagonal of 1s against a background of 0s and then, after reading all memory cells, it shifts the diagonal horizontally. This continues until the diagonal of 1s has passed through all memory locations. The Diagonal test, of order N, will verify address uniqueness at a significant speed enhancement over the Walk or GALPAT.

Checkerboard Test writes 1s and 0s into alternate memory locations in a checkerboard pattern. After a time delay, which may be several seconds, the pattern is read from memory. This pattern is used to evaluate data retention in static RAMs.

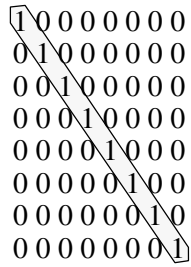


Figure 10.5 The sliding diagonal test.

Surround Read Disturb starts by creating a background of all 0s. Then, each cell in turn becomes the test cell. The test cell is complemented and the eight physically adjacent cells are repeatedly read. After a number of iterations the test cell is read to determine if it has been affected by the read of its neighbors. The operation is then repeated for a background of 1s. The intent is to find disturbances caused by adjacent cell operations. Execution time depends on the number of read cycles but is of the order N.

Surround Write Disturb is identical to the Surround Read Disturb except that a write rather than a read is performed.

Write Recovery writes a background of 0s. Then the first cell is established as the test cell. A 1 is written into the second cell and the first (test) cell is read. The second cell is restored to 0 and the test cell is read again. This is repeated for the test cell and every other cell. Every cell then becomes the test cell in turn. The entire process is repeated using complemented data. This is an N^2 test that is directed at write recovery type faults. It also detects faults that are detected by GALPAT.

Address Test writes a unique value into each memory location. Typically, this could be the address of that memory cell; that is, the value n is written into memory location n. After writing all memory locations, the data are read back. The purpose of this test is to check for address uniqueness. This algorithm requires that the number of bits in each memory word equal or exceed the number of address bits.

Moving Inversions test[4] inverts a memory filled with 0s to 1s and conversely. After initially filling the memory with 0s, a word is read. Then a single bit is changed to a 1, and the word is read again. This is repeated until all bits in the word are set to 1 and then repeated for every word in memory. The operation is then reversed, setting bits to 0 and working from high memory to low memory.

For a memory with n address bits the process is repeated n times. However, on each repetition, a different bit of the address is taken as the least significant bit for incrementing through all possible addresses. An overflow generates an end around carry so all addresses are generated but the method increments through addresses by 1s, 2s, 4s, and so on. For example, on the second time through, bit 1 (when regarding bit 0 as least significant bit, LSB) is treated as the LSB so all even addresses are generated out to the end of memory. After incrementing to address 111...110, the next address generated is address 000...001, and then all consecutive odd addresses are

generated out to the end of memory. The pattern of memory address generation (read the addresses vertically) for the second iteration is as follows:

```
0000 . . . 1111
     . . .
     . . .
     . . .
0000 . . . 1111
0011 . . . 0011
0101 . . . 0101
0000 . . . 1111
```

The Moving Inversions test pattern has $12BN1\log_2 N$ patterns, where B is the number of bits in a memory word. It detects addressing failures and cell opens and shorts. It is also effective for checking access times.

10.4 MEMORY FAULTS

As memories grow larger, with more memory cells packed into an ever-shrinking die area, the cost to manufacture a die remains fairly constant, while the time it takes to apply test programs increases exponentially. It is variously estimated that the cost to test a memory chip runs from 50% to 70% of the total cost of the finished product.[5] The first step in reducing the cost of memory test is to understand what fault mechanisms are most likely to occur and then develop test programs that target those faults. With this approach, the manufacturer and the end-user can determine their priorities, balancing cost versus DPM (defects per million) that they can tolerate in their applications.

A number of different failure types can occur in semiconductor memories, affecting memory cell contents, cell addressing, and the time required to read out data. Some of the more common failures include the following:[6]

Cell opens or shorts
Address nonuniqueness
Cell/column/row disturb sensitivity
Sense amplifier interaction
Slow access time
Slow write recovery
Data sensitivity
Refresh sensitivity
Static data losses

Opens and shorts within semiconductor memory cells may occur because of faulty processing, including misaligned masks or imperfect metallization. These

failures are characterized by a general randomness in their nature. Opens and shorts may occur at the chip connections to a printed circuit board. In a $km \times n$ memory system containing km words of n bits each, and made up of memory chips of size $m \times 1$, a fault that occurs in bit position i of m consecutive bits is indicative of either a totally failed chip or one in which an open or short exists between the chip and the PCB on which it is mounted.

Address nonuniqueness results from address decoder failures that may either cause the same memory cell to be accessed by several different addresses or several cells may be addressed during a single access. These failures often cause some cells to be physically inaccessible. An effective test must insure that each read or write operation accesses one, and only one, memory cell.

Disturb sensitivity between adjacent cells or between cells in the same row or column can result from capacitive coupling. Slow access time can be caused by slow decoders, overloaded sense amplifiers, or an excessive capacitive charge on output circuits. Slow write recovery may indicate a saturated sense amplifier that cannot recover from a write operation in time to perform a subsequent read operation.

A memory cell can be affected by the contents of neighboring cells. Worse still, the cell may be affected only by particular combinations on neighboring cells. This problem grows more serious as the distance between neighboring cells shrinks. Refresh sensitivity in dynamic RAMs may be induced by a combination of data sensitivity and temperature or voltage fluctuations. Static RAM cells are normally able to retain their state indefinitely. However, data may become lost due to leakage current or opens in resistors or feedback paths.

Recall from Section 3.4, when discussing faults in random logic, that fault models other than the stuck-at model were examined. The one trait these models had in common was a susceptibility to combinatorial explosion. For very small circuits, the number of faults grew so quickly that it was simply not feasible to consider them. Memory circuits, because of their density and the close proximity of cells to one another, exhibit this problem of combinatorial explosion to a far greater degree. Hence, it becomes necessary to restrict consideration to faults that are most likely to occur.

The first step is to group the faults into three broad categories: address decoder faults, memory array faults, and read/write logic faults. From there we use the fact, demonstrated by Nair, Thatte, and Abraham,[7] that faults in memory addressing and read/write logic, which includes sense amplifiers, write drivers, and other supporting logic, can be mapped onto functionally equivalent faults in the memory array. This makes it possible to concentrate on faults in the memory array and to develop tests addressed at the functionality of the memory array.

First consider faults in the address decode logic. A fault may cause multiple cells to be accessed, or no cell may be accessed, or the wrong cell may be addressed. In the case of multiple cells being addressed, the fault may be viewed as a coupling fault between cells. If no cell is addressed, then, depending on the logic, the response from the read logic may appear as a stuck-at-1 or a stuck-at-0. If the wrong cell is addressed, then, given the presence of the opposite value in that cell, it appears as a stuck-at fault.

A fault in the read/write logic may cause an output line to be stuck-at-0 or stuck-at-1. In either case, the corresponding cell may be considered to be stuck-at-0 or stuck-at-1. If there are shorts or capacitive coupling between data input or data output lines, these faults can be regarded as coupling between memory cells.

Three conditions, listed below, are defined by Nair et al. in order to detect the faults in their fault model. In the conditions, a *forced transition* is one that occurs as a result of the test algorithm writing into a cell.

Condition 1. Every cell must undergo each of the following two transitions, and must be read after each transition, before undergoing any subsequent forced transitions.

(a) a 0–1 transition and
(b) a 1–0 transition.

Condition 2. For every pair of cells (i, j), cell i must be read after cell j makes a forced transition and before cells i and j make any further forced transitions for the following states of cell i and transitions in cell j:

(a) cell i in state 0, cell j making a 0–1 transition,
(b) cell i in state 1, cell j making a 0–1 transition,
(c) repeat a and b with cell j making a 1–0 transition.

Condition 3. For every cell triple (i, j, k), and $x, y, z \in \{0, 1\}$, if the test makes a transition in cell j from y to $\bar{y}$ after cell i makes a transition from x to $\bar{x}$ and before cell k in state z is read, then the test must possess another sequence where either:

(a) cell k in state z is read after an x to $\bar{x}$ transition in cell i and before a y to $\bar{y}$ transition in cell j, or
(b) cell k in state $\bar{z}$ is read after a y to $\bar{y}$ transition in cell j and before an x to $\bar{x}$ transition in cell i.

Theorem 10.1 The conditions 1 through 3 are necessary and sufficient for a test to detect all the stuck-at and coupling faults in the memory array.

The proof of this theorem is left as an exercise for the reader. An algorithm is now presented that addresses the faults in the fault model just described. The notation used here is from van de Goor.[8]

$[\Uparrow(W0)]$	Initialize to all 0s
$[\Uparrow(R0,W1); \Downarrow(R1)]$	Sequence 1
$[\Uparrow(R1,W0); \Downarrow(R0)]$	Sequence 2
$[\Downarrow(R0,W1); \Uparrow(R1)]$	Sequence 3
$[\Downarrow(R1,W0); \Uparrow(R0)]$	Sequence 4

[⇑(R0,W1,W0); ⇓(R0)]	Sequence 5
[⇓(R0,W1,W0); ⇑(R0)]	Sequence 6
[⇑(W1)]	Initialize to all 1s
[⇑(R1,W0,W1); ⇓(R1)]	Sequence 7
[⇓(R1,W0,W1); ⇑(R1)]	Sequence 8

Legend:

⇑—operate in ascending order	W0—write 0	W1—write 1
⇓—operate in descending order	R0—read 0	R1—read 1

Theorem 10.2 The above algorithm is a complete test for the stated memory fault model.

The algorithm described above is of order n, denoted $O(n)$, where n is the size of memory. In fact, there are 30 read and write passes through memory, so the algorithm is frequently described as being of complexity $30n$. In their paper, Nair et al. point out that the GALPAT, which is $O(n^2)$, does not satisfy all of condition 2. They then define a more comprehensive fault model that includes coupling faults, and they extend the above algorithm to address those faults.

10.5 MEMORY SELF-TEST

Memory ICs keep growing larger. The cost of manufacturing these ICs remains relatively constant as they grow larger, but the cost of testing them increases, because every cell has to be tested for several different kinds of fault mechanisms. It was pointed out in the previous section that the cost of testing these large memory ICs often takes up more than half of the total manufacturing cost.

One of the major contributors to test cost for memory ICs is the commercial testers that are used to test them. In addition to the growing size of memories, the speed at which they operate also continues to increase. In order to keep up with memory IC technology, vendors must constantly upgrade their testers, with a resultant increase in cost. Another problem that must be faced is the inability to access many of the memories because they are embedded in ICs, surrounded by random logic. Gaining access to the address, data and control pins and controlling them with dedicated memory test algorithms is often impossible.

As a result of these growing difficulties, memory built-in self-test (MBIST) has become an accepted way to test many memories. BIST not only has the ability to access embedded memories, but it also has the advantage that it can be designed in conjunction with the memory. Thus, architectural features can be incorporated into the design to take advantage of the presence of BIST. This includes partitioning an internal memory array into several smaller arrays that can be tested in parallel, thus significantly reducing total test time. Note, however, that fragmentation of memories may be a

disadvantage. If a design contains many small memories, the overhead of BIST may be prohibitive.

Another advantage of BIST is its ability to test memory within a system while it is in operation; hence whenever the system is powered up, the memory can be tested for defects that may have occurred since the system was last in operation. This is critical in systems that must be failsafe, particularly as there is some concern that with technology approaching 0.1 micron; soft errors and noise may become major problems.[9]

10.5.1 A GALPAT Implementation

A generic BIST circuit is depicted in Figure 10.6.[10] For memory test the CUT would be the memory module. The BIST must not only generate the data, but must also contain circuits to generate memory addresses in some predetermined order. With minor modifications to the diagram, the same test generator could be used to generate the expected response, by way of the control logic, in addition to the test pattern sequence. In fact, the test generator could generate data to first fill all of memory with some desired pattern, then the same test generator could generate the expected response simultaneously with reading memory. Depending on the algorithm implemented by the test generator, this BIST will test for addressing faults as well as memory faults.

The following synthesizable Verilog code implements a BIST circuit, together with testbench, to perform a GALPAT test. Note that the GALPAT example in Section 10.3 was executed by the testbench; it would be analogous to application of GALPAT from a memory tester. The reader can experiment with the parameters to see how the circuit behaves with different memory sizes. The circuit is easily modified to perform one of several other memory test algorithms that we will discuss subsequently (see the exercises at the end of the chapter). The RAM module, not listed here, is the same one used previously (Section 10.3). It is easily altered to model various fault mechanisms.

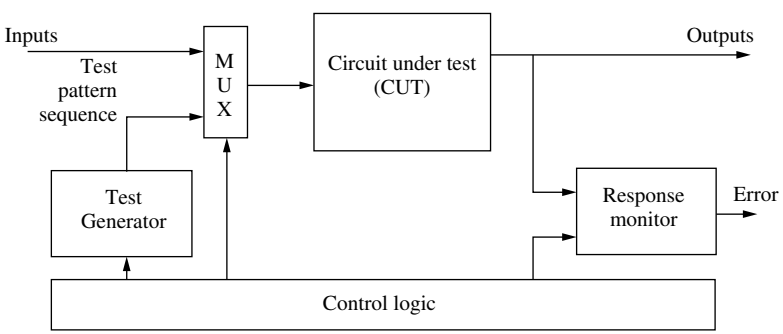


Figure 10.6 Generic BIST scheme.

```
`timescale 1ns / 100ps
module DFF(Q, CLK, set);
input CLK, set;
output Q;
reg Q;
  always @(posedge CLK or posedge set)
    if(set)       Q = 1'b1;
    else if(CLK)  Q = 1'b0;
endmodule
module testbench;
parameter log2_memdepth = 4, memdepth = 16;
wire [log2_memdepth-1:0] addr;
wire datain, wen, oen, dataout, TC, err_flg;
reg T_start, TCLK;
integer counter;
DFF U1 (mem_err, TCLK, err_flg);
ram #(log2_memdepth, memdepth) U2
  (addr,dataout,datain,wen,oen);
galpat #(log2_memdepth,memdepth) U3
  (addr,dataout,datain,wen,oen,TCLK,TC,err_flg,T_start);
initial begin
  TCLK = 0;  // test clock
  T_start = 0;
  counter = 0;
end
always begin
  #2 T_start = 1;
  #6 TCLK = ~TCLK;
  if(TCLK == 1)
    counter = counter + 1;
end
always @(TC or mem_err)
begin
  if(mem_err)
    $display("Mem. Error at location %d, during clock
            %d\n",
        addr, counter);
  if(TC) begin      // Test Complete
    $display("Algorithm required %d clocks\n", counter);
```

```
    $finish;
  end
end
endmodule
module galpat (adval, testval, memval, wen, oen, TCLK,
           TC, err_flg, T_reset);
parameter log2_memdepth = 8, memdepth = 256;
output [log2_memdepth-1:0] adval;
output testval, wen, oen, err_flg, TC;
input memval, TCLK, T_reset;
reg [log2_memdepth-1:0]  j, testcell;
wire [log2_memdepth-1:0] adval;
reg [2:0] GSTATE, GSTATE_next;
reg TC, e, read, write;
wire oen, wen, err_flg, testval;
`define S0    4'b000
`define S1    4'b001
`define S2    4'b010
`define S3    4'b011
`define S4    4'b100
`define S5    4'b101
`define S6    4'b110
`define S7    4'b111
always @(GSTATE or testcell or j or e or memdepth)
  case(GSTATE)
    `S0: GSTATE_next = (testcell == memdepth - 1)
              ? `S1 : `S0;
    `S1: GSTATE_next = `S2;
    `S2: GSTATE_next = `S3;
    `S3: GSTATE_next = (j == 0) ? `S4 : `S2;
    `S4: GSTATE_next = `S5;
    `S5: if (testcell != memdepth-1)
            GSTATE_next = `S1;
         else GSTATE_next = (e == 0) ? `S6 : `S7;
    `S6: GSTATE_next = `S0;
    `S7: GSTATE_next = `S7;
    default: GSTATE_next = `S7;
  endcase
always @(negedge T_reset or posedge TCLK)
```

```
  if(!T_reset) begin
    e  = 0;
    j  = 0;
    TC = 0;  // test complete if TC == 1
    testcell = 0;
    GSTATE   = `S0;
  end
  else begin
    GSTATE = GSTATE_next;
    case(GSTATE)
      `S0: begin      // write background of e, e ∈ {0,1}
             read  = 1;               // disable read
             write = 0;               // enable write
             testcell = testcell+1;
           end
      `S1: testcell = testcell+1;
      `S2: begin                     // read neighbor
             write = 1;               // inhibit write
             read  = (j == testcell) ? 1 : 0;
           end
      `S3: begin                     // read testcell
             j = j+1;
             read  = (j == testcell) ? 1 : 0;
           end
      `S4: begin                   // restore testcell
             write = 0;               // enable write
             read  = 1;
           end
      `S5: write = 1;                // disable write
      `S6: e  = 1;
      `S7: TC = 1;
    endcase
  end
assign wen = !TCLK | write;
assign oen = !TCLK | read;
assign err_flg = !oen & !read & (memval ^ e);
assign testval = (GSTATE == `S1 || GSTATE == `S3)
           ? !e : e;
assign adval   = (GSTATE == `S2) ? j : testcell;
endmodule
```

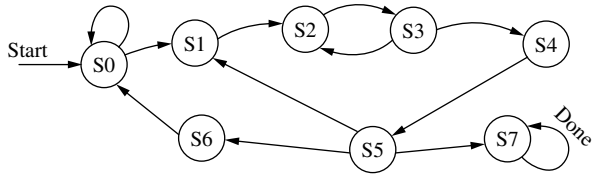


Figure 10.7 State graph for GALPAT.

In this model, corresponding to the state diagram in Figure 10.7, the background is written during state S0. In state S1 the testcell is chosen and set to the value e. Then, during states S2 and S3 the circuit "ping-pongs" back and forth, alternately reading the test cell and one of the neighbor cells. In S4 the testcell is restored. S5 transitions to S1 if additional memory locations remain to act as testcells. If all of the memory locations have served as test cells, then $e \in \{0,1\}$ is checked to determine if both values have been processed. If not, then the state machine transitions from S5 to S6; otherwise it transitions to S7, where it is done.

It is instructive to examine the gate count of this circuit as it is synthesized for various memory sizes. The gate counts will of course vary as a function of the options chosen during synthesis, and those will in turn depend on whether the user chooses to optimize for speed or die area. But, nonetheless, the gate counts vary in proportion to the number of address bits, rather than to memory size.

SYNTHESIS SIZE FOR GALPAT

Memory size	Address bits	Gate Count
16 bits	4	265
64 bits	6	354
256	8	440
64 K	16	760
16 M	24	1090

The GALPAT is impractical, even in BIST form, for all but the smallest memories. The problem is not the gate count but, rather, the execution time. The circuit "ping-pongs" between states S2 and S3 for each testcell. Hence, ignoring the background write in state S0, it is of approximate duration $(2n)^2$, where n is the number of memory cells. This is often expressed as $O(n^2)$, read as "order of n-squared," meaning that computation time is dominated by the square of the number of memory cells. This BIST circuit is easily modified to implement a walking pattern, but the inherent problem of execution time remains.

10.5.2 The 9N and 13N Algorithms

A number of BIST circuits have been proposed in the literature, and in each case the key to successfully defining a memory BIST lies in identifying the fault classes that

are of interest and tailoring an algorithm to address those faults. Then, a hardware implementation, or BIST, circuit can be designed to implement that algorithm. The 9N and 13N algorithms will be described here, where N is the number of memory locations.[11] The 13N has been used in the AMD K6 microprocessor.[12] Implementation of BIST circuits[13] for 9N and 13N is left as an exercise.

Development of the 9N and 13N algorithms began with a study of a number of spot defects in an $8k \times 8$ SRAM memory. The defects were first translated to defects in the circuit transistor diagram. Then, defects at transistor level were classified based on equivalent faulty memory behavior. The result was six fault classes:

1. A cell is stuck-at 0 or stuck-at 1.
2. A cell is stuck-open.
3. A cell has a transition fault.
4. A cell is state coupled to another cell.
5. A multiple access fault exists from one memory cell to another.
6. A cell has a data retention fault.

In their study, the authors found that about 50% of the faults were stuck-at faults. Data retention was the second most common fault, while multiple access faults were least common. The 13N algorithm is used when the SRAM sense amplifiers include a data latch. The purpose of the latch is to extend the read window of the RAM. However, during testing, this latch can mask the effects of stuck-open faults. The 9N algorithm is shown to be sufficient to detect all the faults of interest when there is no data latch.

[⇑(W0)]	Initialize to all 0s
[⇑(R0,W1)]	Sequence 1
[⇑(R1,W0)]	Sequence 2
[⇓(R0,W1)]	Sequence 3
[⇓(R1,W0)]	Sequence 4
Disable RAM	Wait
[⇑(R0,W1)]	Sequence 5
Disable RAM	Wait
[⇑(R1)]	Sequence 6

The first five rows in this algorithm constitute the 9N algorithm. The inclusion of the final four rows extend it to a 13N algorithm. The duration of the wait depends on the node capacitance and leakage current in the memory cells. It is proven in the original paper that all of the fault classes of interest are detected by the 9N and/or the 13N algorithms. We will consider here the proof for detection of coupling faults.

Theorem 10.3 The 9N/13N algorithm detects all state coupling faults.

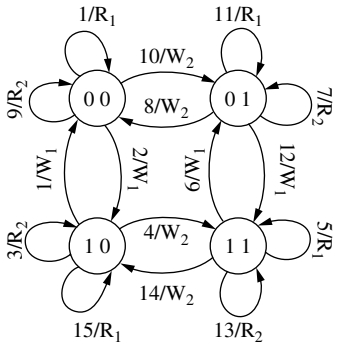


Figure 10.8 Cell checking sequence for 9N algorithm.

Proof The test for state coupling is proven by demonstrating that, for any two arbitrary cells, all four binary combinations exist on these two cells, and are checked, at some point during the test. We start by designating two arbitrary cells to be cell1 and cell2. Then, in the state diagram in Figure 10.8, the binary values inside the state circles represent the values on cell1 and cell2. The arc labeled $1/R_1$ represents step 1, which is a read of cell1. Then, the arc $2/W_1$ represents a transition to the state (1, 0), and represents a write to cell 1. In that state, the arc $3/R_2$ represents a read of cell2. The remaining transitions are interpreted similarly. The reader can confirm that all combinations are checked at some point during the algorithm.

10.5.3 Self-Test for BIST

One of the questions that occasionally comes up concerns failures in the BIST circuits. What happens if the BIST fails? First, it should be considered that, for large memory arrays, the BIST circuitry is a small percentage of the total die area. Consequently, the DPM (defects per million) attributable to the BIST should be very small. One way to further reduce the DPM caused by BIST is to use less aggressive scaling in the BIST circuits so as to realize greater reliability. Another approach that can further reduce the DPM caused by BIST is to incorporate BIST circuits in the BIST. In Section 9.8.2 a self-test feature was described that took advantage of the parity of one-hot encoded state machines. A large percentage of the defects in the state machine are immediately detectable by virtue of the fact that they will cause an even number of flip-flops to be turned on. A parity check on these flip-flops reveals stuck-at faults not only in the flip-flops, but in the logic that controls the state transitions.

10.5.4 Parallel Test for Memories

Conceptually, it is inviting to think of a memory as being composed of a single, monolithic array. This is in part due to the fact that we usually have no visibility into

the inner workings of the memory IC. However, the manufacturer does know intimately the layout of the device. Furthermore, the manufacturer can tailor the architecture of the memory to a specific set of objectives. One of these objectives is the test time. It is possible to divide a memory array into several smaller arrays and test them in parallel. This can lead to a significant reduction in test time. A taxonomy for different memory architectures includes the following:[14]

SASB	Single-array single-bit
SAMB	Single-array multiple bit
MASB	Multiple-array single bit
MAMB	Multiple-array multiple bit

When testing with an external tester the SASB is the prevalent view of the memory DUT, since the tester transmits one address at a time to the IC. The SAMB usually accesses multiple bits from the same row of the DUT. The MAMB accesses multiple bits from multiple arrays and provides the biggest improvement in test time. In a 4-Mb DRAM designed by Toshiba, 16 bits can be tested in parallel.[15] In addition, the chips can be tested in parallel when mounted on a PCB for additional savings in test time.

In yet another parallel test mode, an MISR (cf. Section 9.3) is used as part of a parallel test algorithm.[16] This is illustrated in Figure 10.9. It is not a true BIST scheme since it depends on scan-in of data via a tester and scan-out of the accumulated signature. The scan mechanism is a variant of the BILBO (cf. Section 9.3.4); since it is multipurpose, it can be used to scan data in and out as well as to accumulate signatures.

The discussion here is simplified by assuming that there is a flip-flop in the MISR for every sense amplifier. The designer actually has several options in this scheme. For example, the designer could access the bit line chosen by the column decoder. Also note that the number of flip-flops used in the MISR could exceed the number of bit lines in order to reduce aliasing.

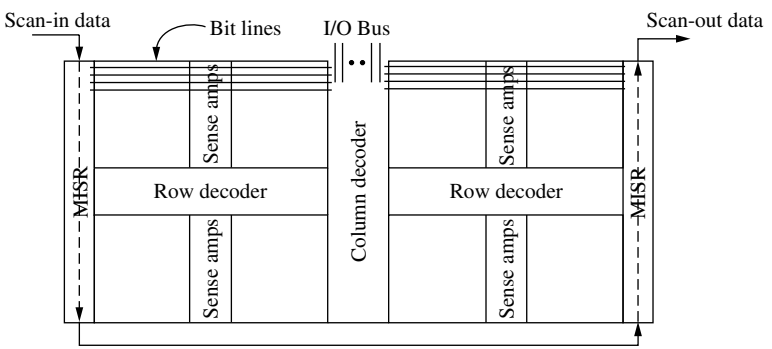


Figure 10.9 MISR used to test memory.

Consider an $N \times 1$ memory with k bit lines accessed by k flip-flops. Define a p-word to be all of the memory locations that can simultaneously be written to or read from by the MISR. A march pattern proceeds as follows:

Step 1: Write a background of zeros.

Step 2: For $i = 1$ to N, do the following:
(a) Read all-zeros from word(i).
(b) Write all-ones into word(i).
(c) Read all-ones from word(i).

Step 3: For $i = 1$ to N, do the following:
(a) Read all-ones from word(i).
(b) Write all-zeros into word(i).
(c) Read all-zeros from word(i).

Step 4: Repeat the sequence with complementary pattern.

For the modified march pattern, using the MISR, the march pattern becomes:

Step 1: Write a background of zeros.
(a) Scan all zero pattern into MISR.
(b) For $i = 1$ to N/k do the following: Write MISR contents into p-word(i).

Step 2: For $i = 1$ to N/k do the following:
(a) Read all-zeros from p-word(i).
(b) {Scan all-one pattern into MISR}—optional. Write MISR contents into p-word(i).
(c) Read all-ones from p-word(i).

Step 3: For $i = 1$ to N/k do the following:
(a) Read all-ones from p-word(i).
(b) Scan all-zero pattern into MISR}—optional. Write MISR contents into p-word(i).
(c) Read all-zeros from p-word(i).

Step 4: Repeat the sequence with complementary pattern.

The original march is a 14N algorithm. The modified march, with the scan-in, will have duration $2[(k + N/k) + N/k(1 + k + 1 + 1) + N/k(1 + k + 1 + 1)]$, which reduces to $2(2N + k + 7N/k)$. Because of all the scan operations, this is not a significant savings. For example, if N = 1Mbit, and k = 1Kbit, the modified march is approximately 4N, which represents a savings of 3.5X. However, if the contents of the MISR are used as stimuli, then the duration of the test is $2[(k + N/k) + \{k + N/k(1 + 1 + 1)\} + \{k + N/k(1 + 1 + 1)\}]$. This reduces to $2(3k + 7N/k)$, or approximately $14N/k$. For the 1Mbit memory, that represents a savings of about 1000X.

10.5.5 Weak Read–Write

One of the more expensive tests, in terms of test time, is the data retention test. Recall, from discussion of the 9N and the 13N tests, that the difference between the

9N and the 13N derived from the fact that 13N included two wait periods. The duration of the wait periods can be considerable, and it represents a substantial cost. The weak write test mode addresses this problem through the use of some built-in test circuits. The object is to try to precipitate early failures of those ICs that would otherwise fail during operation. This is illustrated in Figure 10.10. The SRAM is made up of transistors M1 through M6. The additional circuitry, composed of Ma through Mf, constitutes the weak write test mode (WWTM). During normal operation the weak write circuits are disabled. Since the WWTM is nonintrusive, it has no impact on normal performance of the SRAM.

The operation of the circuits during test is as follows:

1. Write a background of all 0s to the memory array.
2. Enable WWTM.
3. Perform weak write one (WR1).
4. Disable WWTM.
5. Perform a read, determine if any cell has been overwritten.
6. Repeat with complementary values.

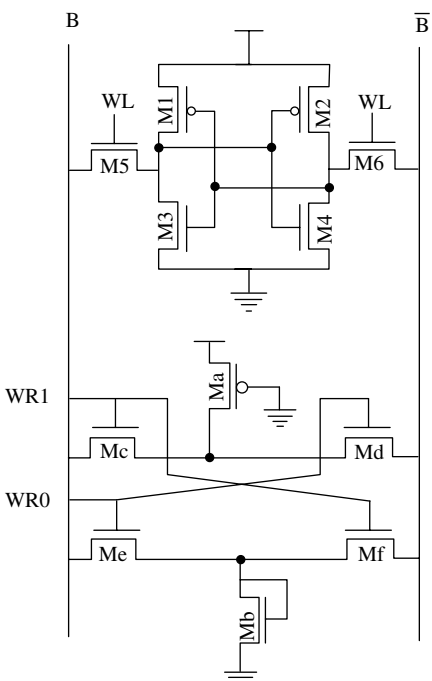


Figure 10.10 SRAM cell with weak write.

The key to operation of WWTM is the use of transistors with size and bias that permit only weak or faulty SRAM cells to be overwritten. As a result, cells that are within specification are unaffected by the weak write circuits.

The amount of die space taken by the weak write circuits depends on how many SRAM cells share the word lines with the WWTM circuit. If a block of memory has 128-word lines, then the additional circuitry contributed by the WWTM is less than 1%. In computing the savings in test time, two tests are considered. The pause test, or data retention test, writes a background to the array and, after a pause of perhaps 100 ms, the array is read to determine if any cell has changed state. The read disturb test writes a background to the array, and then it reads the array at a higher or lower V_{cc} while ignoring the data. After some elapsed time the array is again read to determine if any cell has changed state.

In each of these tests, the elapsed time of the pause is measured in hundreds of milliseconds. WWTM, conversely, required only microseconds. The impact on production test was a savings of 20% in test time, plus the savings realized by not packaging defective die. An additional fallout from WWTM is the detection of faults that are not detected by the previous method.

10.6 REPAIRABLE MEMORIES

With the growing number of cells in each memory IC and with shrinking feature sizes, yield becomes a more critical problem. Defect densities, measured in defects per square centimeter, continue to decrease, as a result of improved processes and cleaner fab rooms, but die size increases as the number of memory cells quadruples from one generation of memories to the next. Furthermore, faulty operation can be caused by mask imperfections or pinhole defects that would not have caused errors in a die with larger feature sizes.

In a die populated with random logic, there is no predictable order to the placement of logic cells, and faulty die are normally discarded. However, since a significant portion of the faulty die contain only a few faulty memory cells,[17] it is possible to take advantage of the regular structure of RAM chips and add extra rows and columns to improve yield. During test, if a memory chip is discovered to have just a few bad cells, then one of the spare row(s) or column(s) is substituted for the faulty row(s) or column(s) containing the bad cell(s) to create a good memory chip.

The substitution of a row or column for another one is achieved by means of fuses. In Figure 10.11 the SRE (spare row enable) signal is normally held high so the output of the spare row NOR decoder is held high. If the spare row is to be used, then an SRE fuse is blown, which enables the spare row NOR. If the spare row NOR is selected, it disables all other NORs. There is a programming element for each row address line. The programming elements (Figure 10.12) determine the row address to be selected. During test, if it is determined that the die can be repaired by substituting a spare row for a row with failed cells, then the SRE signal is activated by blowing a fuse. The address of the failed row is then programmed

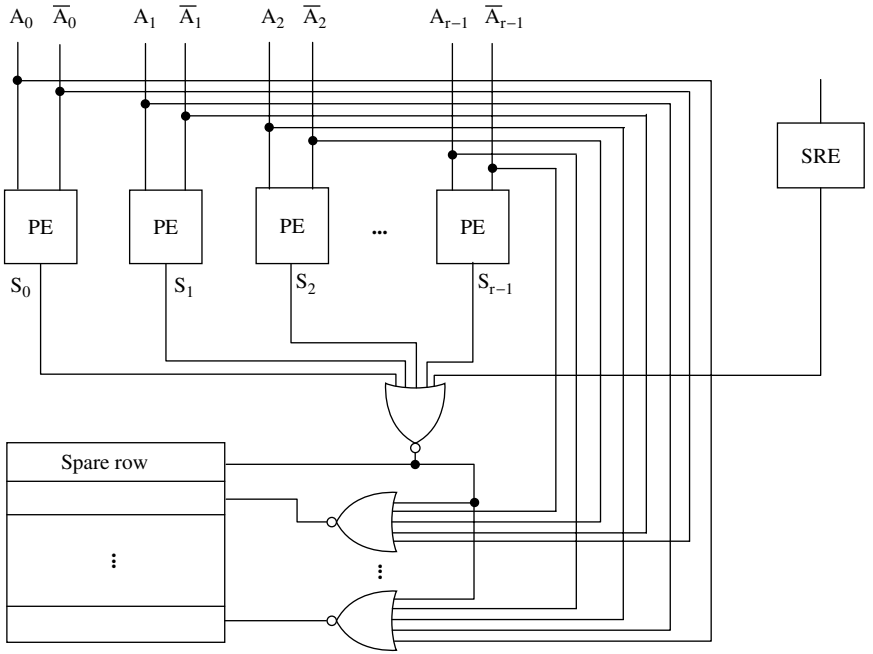


Figure 10.11 Alternate row select.

into the programming elements. If the fuse in the PE is blown, the output S_i is connected to $\overline{A_i}$ because T_1 is enabled through transistor D. If the fuse is not blown, then the transistor T_2 is activated and S_i is connected to A_i.

Each programming element has a unique address. If the address fuse in that PE is addressed, its output enables transistor P and a large current flows through the fuse, causing it to open. These PE address lines and V_{DP} are accessible on the die but are

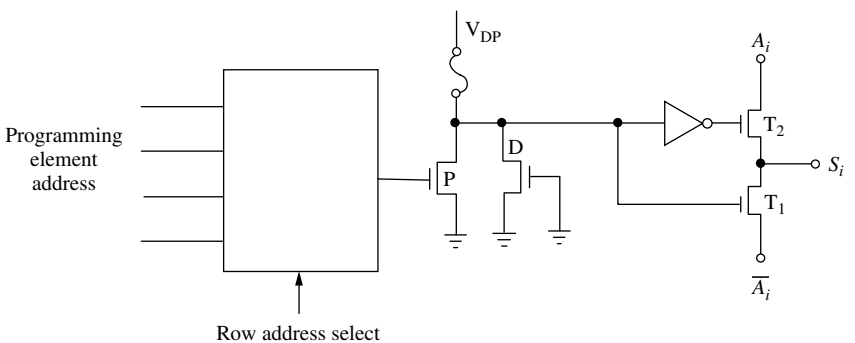


Figure 10.12 Programming element.

not accessible after the chip has been packaged. Each programming element has a unique address. If the address fuse in that PE is addressed, its output enables transistor P and a large current flows through the fuse, causing it to open. These PE address lines and V_{DP} are accessible on the die but are not accessible after the chip has been packaged.

The spare row concept can also be applied to spare column replacement. Furthermore, more than one spare row and column can be provided. Practical considerations usually limit the spares to two rows and two columns, since additional rows and columns cause die size to grow, countering the objective of maximizing yield. When a row or column has replaced another row or column, it is necessary to retest the die to ensure that the substitute row or column is not defective. In addition, it is necessary to verify that the fuse has blown and that the mechanism used to blow the fuse has not caused damage to surrounding circuitry.

There appears to be negligible effect on memory access time due to rows or columns being substituted. The presence of the additional transistor, $T1$ or $T2$, causes roughly an 8% increase in access time. An area of concern with redundant rows and columns is the effect on those memory tests intended to uncover disturb sensitivities. However, comparison of test data between devices with and without redundancies showed no significant differences.[17]

10.7 ERROR CORRECTING CODES

Because of shrinking cell size, semiconductor memories are increasingly prone to random or intermittent errors. These soft errors may be caused by noise, capacitance, or alpha particles. The alpha particles are helium nuclei resulting from decay of radioactive elements in the packaging material.The term *soft error* refers to the fact that the error is not easily repeatable and the conditions leading up to its occurrence cannot be duplicated. Therefore a specific test to detect its presence does not exist, in contrast to permanent or hard errors for which tests can be created. Soft errors can be dealt with by means of error correcting codes (ECC), also called error detection and correction codes (EDAC). We will look at hard faults, tests devised to detect these faults, and error correcting codes used to recover from the effects of soft errors.

In 1948 Claude Shannon published his classic article entitled "The Mathematical Theory of Communication."[18] In that paper he proved the following theorem:

Theorem 10.4 Let a source have entropy H (bits per symbol) and let a channel have a capacity C (bits per second). Then it is possible to encode the output of the source in such a way as to transmit at the average rate (C/H)-e symbols per second over the channel where e is arbitrarily small. It is not possible to transmit at an average rate greater than C/H.

This theorem asserts the existence of codes which permit transmission of data through a noisy medium with arbitrarily small error rate at the receiver. The alternative, when transmitting through a noisy medium, is to increase transmission power

to overcome the effects of noise. An important problem in data transmission is to minimize the frequency of occurrence of errors at a receiver with the most economical mix of transmitter power and data encoding.

An analogous situation exists with semiconductor memories. They continue to shrink in size; hence error rates increase due to adjacent cell disturbance caused by the close proximity of cells to one another. Errors can also be caused by reduced charge densities.[19] Helium nuclei from impurities found in the semiconductor packaging materials can migrate toward the charge area and neutralize enough of the charge in a memory cell to cause a logic 1 to be changed to a 0. These soft errors, intermittent in nature, are growing more prevalent as chip densities increase. One solution is to employ a parity bit with each memory word to aid in the detection of memory bit errors. A single parity bit can detect any odd number of bit errors. Detection of a parity error, if caused by a soft error, may necessitate reloading of a program and/or data area.

If memory errors entail serious consequences, the alternatives are to use more reliable memories, employ error correcting codes, or possibly use some combination of the two to reach a desired level of reliability at an acceptable cost. Since Shannon's article was published, many families of error correcting codes have been discovered. In memory systems the Hamming codes have proven to be popular.

10.7.1 Vector Spaces

An understanding of Hamming Codes requires an understanding of vector spaces, so we introduce some definitions. A *vector* is an ordered n-tuple containing n elements called scalars. In this discussion, the scalars will be restricted to the values 0 and 1. Addition of two vectors is on an element-by-element basis, for example,

$$v_1 + v_2 = (v_{11}, v_{12}, \ldots, v_{1n}) + (v_{21}, v_{22}, \ldots, v_{2n}) = (v_{11} + v_{21}, v_{12} + v_{22}, \ldots, v_{1n} + v_{2n})$$

The addition operation, denoted by +, is the mod 2 operation (exclusive-OR) in which carries are ignored.

Example If $v_1 = (0, 1, 1, 0)$ and $v_2 = (1, 1, 0, 0)$,
then $v_1 + v_2 = (0 + 1, 1 + 1, 1 + 0, 0 + 0) = (1, 0, 1, 0)$. ■■

Multiplication of a scalar $a \in \{0, 1\}$ and a vector v_1 is defined by

$$av_1 = (av_{11}, av_{12}, \ldots, av_{1n})$$

The *inner product* of two vectors v_1 and v_2 is defined as

$$\begin{aligned} v_1 \cdot v_2 &= (v_{11}, v_{12}, \ldots, v_{1n}) \cdot (v_{21}, v_{22}, \ldots, v_{2n}) \\ &= (v_{11} \cdot v_{21} + v_{12} \cdot v_{22} + \cdots + v_{1n} \cdot v_{2n}) \end{aligned}$$

If the inner product of two vectors is 0, they are said to be *orthogonal*.

A *vector space* is a set V of vectors which satisfy the property that all linear combinations of vectors contained in V are themselves contained in V, where the linear combination u of the vectors $v_1, v_2, ..., v_n$ is defined as

$$u = a_1 v_1 + a_2 v_2 + \cdots + a_n v_n \qquad a \in \{0, 1\}$$

The following additional properties must be satisfied by a vector space:

1. If $v_1, v_2 \in V$, then $v_1 + v_2 \in V$.
2. $(v_1 + v_2) + v_3 = v_1 + (v_2 + v_3)$.
3. $v_1 + e = v_1$ for some $e \in V$.
4. For $v_1 \in V$, there exists v_2 such that $v_1 + v_2 = e$.
5. The product $a \cdot v_1$ is defined for all $v_1 \in V$, $a \in \{0,1\}$.
6. $a(v_1 + v_2) = av_1 + av_2$.
7. $(a + b)v_1 = av_1 + bv_1$.
8. $(ab)v_1 = a(bv_1)$.

A set of vectors $v_1, v_2, ..., v_n$ is *linearly dependent* if there exist scalars $c_1, c_2, ..., c_n$, not all zero, such that

$$c_1 v_1 + c_2 v_2 + \cdots + c_n v_n = 0$$

If the vectors $v_1, v_2, ..., v_n$ are not linearly dependent, then they are said to be *linearly independent*.

Given a set of vectors S contained in V, the set $L(S)$ of all linear combinations of vectors of S is called the linear span of S. If the set of vectors S is linearly independent, and if $L(S) = V$, then the set S is a *basis* of V. The number of vectors in S is called the *dimension* of V.

A subset U contained in V is a subspace of V if $u_1, u_2 \in U$ implies that

$$c_1 v_1 + c_2 v_2 \in U \text{ for } c_1, c_2 \in \{0, 1\}.$$

The following four theorems follow from the above definitions:

Theorem 10.5 The set of all n-tuples orthogonal to a subspace V_1 of n-tuples forms a subspace V_2 of n-tuples. This subspace V_2 is called the *null space* of V_1.

Theorem 10.6 If a vector is orthogonal to every vector of a set which spans V_1, it is in the null space of V_1.

Theorem 10.7 If the dimension of a subspace of n-tuples is k, the dimension of the null space is $n - k$.

Theorem 10.8 If V_2 is a subspace of n-tuples and V_1 is the null space of V_2, then V_2 is the null space of V_1.

Example The vectors in the following matrix, called the *generator matrix* of *V*, are linearly independent. They form a basis for a vector space of 16 elements.

$$G = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{bmatrix} \tag{10.1}$$

The dimension of the subspace defined by the vectors is 4. The vectors 0111100, 1011010, and 1101001 are orthogonal to all of the vectors in G, hence they are in the null space of G. Furthermore, they are linearly independent, so they define the following generator matrix H for the null space of V:

$$H = \begin{bmatrix} 0 & 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 & 1 \end{bmatrix} \tag{10.2}$$

■■

10.7.2 The Hamming Codes

From Theorem 10.8 we see that a vector space can be defined in terms of its generator matrix G or in terms of the generator matrix H for its null space. Since a vector $v \in V$ is orthogonal to every vector in the null space, it follows that

$$v \cdot H^T = 0 \tag{10.3}$$

where H^T is the transpose of H.

The *Hamming weight* of a vector v is defined as the number of nonzero components in the vector. The *Hamming distance* between two vectors is the number of positions in which they differ. In the vector space generated by the matrix G in Eq. (10.1), the nonzero vectors all have Hamming weights equal to or greater than three. This follows from Eq. 10.3, where the vector v selects columns of H which sum, mod 2, to the 0 vector. Since no column of H contains all zeros, and no two columns are identical, v must select at least three columns of H in order to sum to the 0 vector.

Let a set of binary information bits be represented by the vector $J = (j_1, j_2, \ldots, j_k)$. If G is a $k \times n$ matrix, then the product $J \cdot G$ encodes the information bits by selecting and creating linear combinations of rows of G corresponding to nonzero elements in J. Each information vector is mapped into a unique vector in the space V defined by the generator matrix G. Furthermore, if the columns of the generator matrix H of the null space are all nonzero and if no two columns of H are identical, then the encoding produces code words with minimum Hamming weight equal to 3. Since the sum of any two vectors is also contained in the space, the Hamming distance between any two vectors must be at least three. Therefore, if one or two bits are in error, it is possible to detect the fact that the encoded word has been altered.

If we represent an encoded vector as v and an error vector as e, then

$$(v + e) \cdot H^T = v \cdot H^T + e \cdot H^T = eH^T$$

If e represents a single bit error, then the product eH^T matches the column of H corresponding to the bit in e which is nonzero.

Example If G is the matrix in Eq. (10.1), and $J = (1,0,1,0)$, then $v = J \cdot G = (1,0,1,0,1,0,1)$. If $e = (0,0,0,1,0,0,0)$, then $v + e = (1,0,1,1,1,0,1)$. So,

$$v + eH^T = (1, 0, 1, 1, 1, 0, 1) \begin{bmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = (1, 1, 1)$$

The product (1,1,1) matches the fourth column of H (fourth row of H^T). This implies that the fourth bit of the message vector is in error. Since the information bits are binary, it is a simple matter to invert the fourth bit to get the original vector (1,0,1,0,1,0,1). ■■

In this encoding the first four columns of G form an identity matrix; hence when we multiply J and G, the first four elements of the resulting vector match the original information vector. Such a code is called a *systematic code*. In general, the columns of G can be permuted so that columns making up the identity matrix can appear anywhere in the matrix. The systematic code is convenient for use with memories since it permits data to be stored in memory exactly as it exists outside memory. A general form for G and H, as systematic codes, is

$$G = [I_k;\ P_{k(n-k)}]$$

$$H = [P^T_{(n-k)k};\ I_{(n-k)}]$$

where I_n is the identity matrix of dimension n, the parameter k represents the number of information bits, n is the number of bits in the encoded vector, and $n - k$ is the number of parity bits. The matrix P is called the parity matrix, the generator matrix H is called the parity check matrix, and the product $v \cdot H^T$ is called the syndrome. When constructing an error correcting code, the parameters n and k must satisfy the expression $2^{n-k} - 1 \geq n$.

Error correcting codes employ maximum likelihood decoding. This simply says that if the syndrome is nonzero, the code vector is mapped into the most likely message vector. In the code described above, if the syndrome is (1,1,1), it is assumed that bit 4 of the vector is in error. But, notice that the 2-bit error $e = (1,0,0,1,0,0)$ could have produced the same syndrome. This can cause a false correction because maximum likelihood decoding assumes that one error is more probable than two

errors; that is, if Pi is the probability that the ith bit is received correctly, then $P_i > Q_i = 1 - P_i$, where Q_i is the probability of receiving the incorrect bit.

To avoid the possibility of an incorrect "correction," an additional bit can be added to the code vectors. This bit is an even parity check on all of the preceding bits. The parity matrix P for the preceding example now becomes

$$P = \begin{bmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{bmatrix}$$

Since the information vectors must now be even parity, any odd number of errors can be detected. The decoding rule is as follows:

1. If the syndrome is 0, assume no error has occurred.
2. If the last bit of the syndrome is one, assume a single-bit error has occurred; the remaining bits of the syndrome will match the column vector in H corresponding to the error.
3. If the last bit of the syndrome is zero, but other syndrome bits are one, an uncorrectable error has occurred.

In case 3, an even number of errors has occurred; consequently it is beyond the correcting capability of the code. An error bit may be set when that situation is detected, or, in a computer memory system, an uncorrectible error may trigger an interrupt so that the operating system can take corrective action.

10.7.3 ECC Implementation

An ECC encoder circuit must create parity check bits based on the information bits to be encoded and the generator matrix G to be implemented. Consider the information vector $J = (j_1, j_2, \ldots, j_k)$ and $G = [I_k; P_{k \times r}]$, where $r = n - k$ and

$$P_{k \cdot r} = \begin{bmatrix} p_{11} & p_{12} & \cdots & p_{1r} \\ p_{21} & p_{22} & \cdots & p_{2r} \\ \cdots & \cdots & \cdots & \cdots \\ p_{k1} & p_{k2} & \cdots & p_{kr} \end{bmatrix}$$

In the product $J \cdot G$, the first k bits remain unchanged. However, the $(k + s)$th bit, $1 \le s \le r$, becomes

$$\begin{aligned} g_s &= j_1 \cdot p_{1s} + j_2 \cdot p_{2s} + \cdots + j_k \cdot p_{ks} \\ &= \sum_{m=1}^{k} j_m \cdot p_{ms} \end{aligned}$$

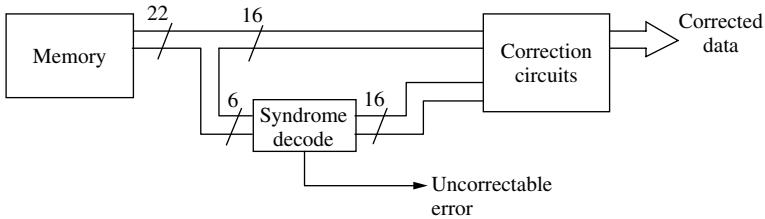


Figure 10.13 Error correction circuit.

Therefore, in an implementation, the $(k + s)$th symbol is a parity check on information bits corresponding to nonzero elements in the sth column of P.

The encoded vector is decoded by multiplying it with the parity generator H to compute the syndrome. This gives

$$(v + e) \cdot H^T = (v_1, v_2, \ldots, v_n) \cdot \begin{bmatrix} P_{k \cdot r} \\ I_r \end{bmatrix} + e \cdot \begin{bmatrix} P_{k \cdot r} \\ I_r \end{bmatrix}$$

$$= (j_1, j_2, \ldots, j_k, p_1, p_2, \ldots, p_r) \cdot \begin{bmatrix} P_{k \cdot r} \\ I_r \end{bmatrix} + e \cdot \begin{bmatrix} P_{k \cdot r} \\ I_r \end{bmatrix}$$

Therefore, to decode the vector, encode the information bits as before, and then exclusive-OR them with the parity bits to produce a syndrome. Use the syndrome to correct the data bits. If the syndrome is 0, no corrective action is required. If the error is correctible, use the syndrome with a decoder to select the data bit that is to be inverted. The correction circuit is illustrated in Figure 10.13. With suitable control circuitry, the same syndrome generator can be used to generate the syndrome bits.

Error correcting codes have been designed into memory systems with word widths as wide as 64 bits[20] and have been designed into 4-bit wide memories and implemented directly on-chip.[21] Since the number of additional bits in a SEC-DED Hamming code with a 2^n bit word is $n + 2$, the additional bits as a percentage of data word width decrease with increasing memory width. For a 4-bit memory, 3 bits are needed for SEC and 4 bits for SEC-DED. A 64-bit memory requires 7 bits for SEC and 8 bits for SEC-DED.

10.7.4 Reliability Improvements

The improvement in memory reliability provided by ECCs can be expressed as the ratio of the probability of a single error in a memory system without ECC to the probability of a double error in a memory with ECC.[22] Let $R = e^{-\lambda t}$ be the probability of a single memory device operating correctly where λ is the failure rate of a single memory device. Then, the probability of the device failing is

$$Q = 1 - R = 1 - e^{-\lambda t}$$

Given m devices, the binomial expansion yields

$$(Q + R)^m = R^m + mR^{m-1}Q + \cdots + Q^m$$

Hence, the probability of all devices operating correctly in a memory with $m + k$ bits is R^m, the probability of one failure is $P_1 = mR^{m-1}Q$, and the probability of two errors is

$$P_2 = \frac{(m+k)(m+k-1)R^{m+k-2}}{2}(1-R)^2$$

The improvement ratio is

$$R_i = \frac{P_1}{P_2} = \frac{2m}{(m+k)(m+k-1)} \times \frac{1}{R^{k-1}(1-R)}$$

Example Using a SEC-DED for a memory of 32-bit width requires 7 parity bits. If $\lambda = 0.1\%$ per thousand hours, then after 1000 hours we have

$$R = 0.9990005$$
$$1 - R = 0.0009995$$
$$R_i = \frac{2 \times 32}{39 \times 38} \times \frac{1}{0.9940 \times 0.0009995} = 43.5$$ ■■

The reliability at $t = 10{,}000$ hours is $R_i = 3.5$. This is interpreted to mean that the likelihood of a single chip failure increases with time. Therefore the likelihood of a second, uncorrectable error increases with time. Consequently, maintenance intervals should be scheduled to locate and replace failed devices in order to hold the reliability at an acceptable level. Also note that reliability is inversely proportional to memory word width. As word size increases, the number of parity bits as a percentage of memory decreases, hence reliability also decreases.

The equations for reliability improvement were developed for the case of permanent bit-line failures; that is, the bit position fails for every word of memory where it is assumed that one chip contains bit i for every word of memory. Data on 4K RAMS show that 75–80% of the RAM failures are single-bit errors.[23] Other errors, such as row or column failure, may also affect only part of a memory chip. In the case of soft errors or partial chip failure, the probability of a second failure in conjunction with the first is more remote. The reliability improvement figures may therefore be regarded as lower bounds on reliability improvement.

When should ECC be employed? The answer to this question depends on the application and the extent to which it can tolerate memory bit failures. ECC requires extra memory bits and logic and introduces extra delay in a memory cycle; furthermore, it is not a cure for all memory problems since it cannot correct address line failures and, in memories where data can be stored as bytes or half-words, use of ECC can complicate data storage circuits. Therefore, it should not be used unless a

clear-cut need has been established. To determine the frequency of errors, the mean time between failures (MTBF) can be used. The equation is

$$\text{MTBF} = 1/d\lambda$$

where λ is again the failure rate and d is the number of devices. Reliability numbers for MTBF for a single memory chip depend on the technology and the memory size, but may lie in the range of 0.01–0.2% per thousand hours. A $64\text{K} \times 8$ memory using eight 64K RAM chips with 0.1% per thousand hours would have an MTBF of 125,000 hours. A much larger memory, such as one megaword, 32 bits/word, using the same chips would have an MTBF of 2000 hours, or about 80 days between hard failures. Such failure rates may be acceptible, but the frequency of occurrence of soft errors may still be intolerable.

Other factors may also make ECC attractive. For example, on a board populated with many chips, the probability of an open or short between two IC pins increases. ECC can protect against many of those errors. If memory is on a separate board from the CPU, it may be a good practice to put the ECC circuits on the CPU board so that errors resulting from bus problems, including noise pickup and open or high resistance contacts, can be corrected. A drawback to this approach is the fact that the bus width must be expanded to accomodate the ECC parity bits.

It is possible to achieve error correction beyond the number of errors predicted to be correctable by the minimum distance. Suppose hard errors are logged as they are detected. Then, if a double error is detected and if one of the two errors had been previously detected and logged in a register, the effects of that error can be removed from the syndrome corresponding to the double error to create a syndrome for the error that had not been previously detected. Then, the syndrome for the remaining error can be used to correct for its effect.

Another technique that can be used when a double error is detected is to complement the word readout of memory and store that complement back into memory. Then read the complemented word. The bit positions corresponding to hard cell failures will be the same, but bits from properly functioning cells will be complemented. Therefore, exclusive-OR the data word and its complement to locate the failed cells, correct the word, and then store the corrected word back in memory. This will not work if two soft errors occurred; at least one of the two errors must be a hard error.[24] This technique can also be used in conjunction with a parity bit to correct hard errors.[25] In either case, whether a single-bit parity error or a double error is detected by ECC, the correction procedure can be implemented by having the memory system generate an interrupt whenever an uncorrectable error occurs. A recovery routine residing either in the Operating System or in microcode can then be activated to correct bit positions corresponding to hard errors.

10.7.5 Iterated Codes

The use of parity bits on rows and columns of magnetic tapes (Figure 10.14) constitutes a SEC-DEC code.[26] The minimum Hamming weight of the information plus

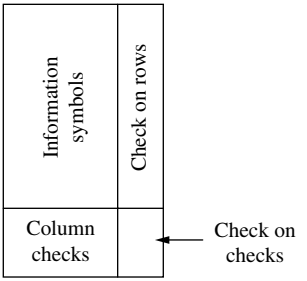


Figure 10.14 Magnetic tape with check bits.

check bits will always be at least 4. In addition, a single-bit error in any position complements a row parity bit, a column parity bit, and the check-on-checks parity bit. Therefore, it is possible to correct single-bit errors and detect double-bit errors.

10.8 SUMMARY

Memories must be tested for functional faults, including cells stuck-at-1 or stuck-at-0, addressing failures, and read or write activities that disturb other cells. Memories must also be tested for dynamic faults that may result in excessive delay in performing a read or write. The cost of testing memory chips increases because every cell must be tested. Some economies of scale can be realized by testing many chips simultaneously on the tester. However, much of the savings in test time over the years has been realized by investigating the fault classes of interest and creating Pareto charts (cf. Section 6.7) to prioritize the failure mechanisms and address those deemed to be most significant. With that information, a test algorithm can be adapted that brings outgoing quality level to acceptable levels.

With feature sizes shrinking, the industry has by and large migrated from core-limited die to pad-limited die. One consequence of this is that BIST represents an insignificant amount of die area relative to the benefit in cost savings, both in time required to test the memory and in the cost of the tester used for that purpose. Just about any test algorithm can be expressed in an HDL such as Verilog or VHDL and synthesized, with the resulting BIST circuit representing perhaps 1.0–2.0% of the die area. Microprogrammed implementations of BIST have also appeared in the literature.[27] A possible advantage of the microprogrammed implementation is that it can be reprogrammed if fault mechanisms change over the life of the chip.

BIST circuits are not only useful during initial fabrication of the die, but they also can be custom tailored for use in everyday operation so that if a defect has occurred while a device is in operation, potentially catastrophic effects on program and/or data can be prevented by running an online test. Transparent BIST can be used as part of an online test.[28] In this mode of operation an online test is run while the device is in operation, but the transparent BIST preserved the contents of memory.

With increasing numbers of memory cells per IC, as well as smaller feature sizes, the possibility of failure, both hard and soft, increases. When failure is

detected during wafer processing, it is possible to substitute another row and/or column for the row or column in which the failure occurred if spare rows and/or columns are provided. This can substantially improve yield, since most of the defective die incur defects in very few rows or colums, hence are repairable.

Recovery from errors during operation can be achieved through the use of ECCs. Analysis of the problem indicates that significant inprovements in reliability can be achieved with the use of ECC. The problem of soft errors was once diagnosed as being caused by radioactive materials in the chip packaging. However, with smaller cells, packed closer together and operating at lower voltages, it can be expected that ECC will regain its popularity.

Finally, we note that the subject of memory design and test is both complex and expanding in scope. This was illustrated by the diagram in Figure 10.1, where virtually every block in that diagram contained some kind of memory. New modes of memory storage constantly appear and existing memories continue to push the technology envelope. It is only possible to briefly cover the existing spectrum of memory devices, with an emphasis on the theoretical underpinnings. The reader desiring to pursue this subject in greater detail is referred to the texts by Prince[29] and van de Goor.[30]

PROBLEMS

10.1 Modify the memory test program of Section 10.3 to implement the following test algorithms:

Galloping Diagonal

Checkerboard

Moving Inversions

10.2 A register set with 16 registers has an SA0 fault on address line A_2 (there are four lines, $A_0 - A_3$). Pick any memory test algorithm that can detect addressing errors and explain, in detail, how it will detect the fault on A_2.

10.3 Using your favorite HDL language/simulator, alter the galpat.v module in Section 10.5.1 to implement the following algorithms: walking, sliding, 9N, 13N.

Run simulations for various memory sizes and, using the counter in the testbench, plot the number of clock cycles versus memory size.

10.4 Synthesize the BIST circuits created in the previous problem. For several sizes of the parameters, plot the gate count versus memory size.

10.5 Insert various faults in the RAM model of Section 10.3, including SA0 and SA1, short to neighbor, addressing faults, and so on, and note which memory tests detect the injected faults.

10.6 Remodel the RAM circuit to show more detail—for example, sense amps, RAS, CAS, write lines, bit lines, and so on. Then insert faults that are visible

only at that level of detail. Determine, by means of simulation, which of the memory test algorithms detect the faults.

10.7 Suppose that a particular die is made up of 55% memory and 45% random logic. Assume that in shipped parts, memory has 2 DPM (defects per million) and that the logic has 1100 DPM. What is the overall DPM for the chip? If process yield for the logic is 70%, what fault coverage is needed to have less than 500 DPM for the shipped parts?

10.8 Create the (8,4) SEC-DED matrix for the following generator matrix G.

$$G = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{bmatrix}$$

10.9 Create the parity check matrix H corresponding to the generator matrix G of the previous problem.

10.10 Using the (8,4) parity check matrix H of the previous problem, determine which of the following vectors are code vectors and which have errors that are (a) correctable, (b) detectable.

1 0 0 1 0 0 0 1
0 1 1 1 1 1 1 0
1 0 1 0 1 0 1 0
1 0 1 0 1 1 0 0
1 1 1 0 0 0 0 1
0 1 0 1 1 1 0 1

10.11 If it is known that bit 3 of all the code words has been identified as a solid SA0, use that information and the matrix H previously given to correct the following vectors:

0 1 0 0 0 1 1 1
0 1 0 1 1 0 1 0
1 0 0 1 0 0 1 0

10.12 For an SEC-DED code, the decoding rules were given for three conditions of the syndrome. However, nothing was said about the condition where the last bit of the syndrome is one, but all other bits are 0. What would you do in that case?

10.13 Prove that the inequality $2^{n-k} - 1 \geq n$ must hold for Hamming codes.

10.14 Prove Theorems 10.5 through 10.8.

REFERENCES

1. Pyron, C. et al., Next Generation PowerPC Microprocessor Test Strategy Improvements, *IEEE Int. Test Conf.*, 1997, pp. 414–423.
2. Stolicny, C. et al., Manufacturing Pattern Development for the Alpha 21164 Microprocessor, *Proc. IEEE Int. Test Conf.*, 1997, pp. 278–286.
3. Intel Corp., Product Overview, 1993, pp. 5–12.
4. de Jonge, J. H., and A. J. Smulders, Moving Inversions Test Pattern is Thorough, Yet Speedy, *Comput. Des.*, Vol. 15, No. 5, May 1976, pp. 169–173.
5. van de Goor, A. J., Using March Tests to Test SRAMs, *IEEE Des. Test*, Vol. 10, No. 1, March 1993, pp. 8–14.
6. Application Note, Standard Patterns for Testing Memories, *Electron. Test*, Vol. 4, No. 4, April 1981, pp. 22–24.
7. Nair, J., S. M. Thatte, and J. A. Abraham, Efficient Algorithms for Testing Semiconductor Random-Access Memories, *IEEE Trans. Comput.*, Vol. C-27, No. 6, June 1978, pp. 572–576.
8. van de Goor, A. J., Testing Memories: Advanced Concepts, *Tutorial 12*, International Test Conference, 1997.
9. Panel Discussion, A D&T Roundtable: Online Test, *IEEE Des. Test Comput.*, January–March 1999, Vol. 16, No. 1, pp. 80–86.
10. Al-Assad, H. et al., Online BIST For Embedded Systems, *IEEE Des. Test Comput.*, October–December 1998, Vol. 15, No. 6, pp. 17–24.
11. Dekker, R. et al., Fault Modeling and Test Algorithm Development for Static Random Access Memories, *Proc. Int. Test Conf.*, 1988, pp. 343–352.
12. Fetherston, R. S. et al., Testability Features of AMD-K6 Microprocessor, *Proc. Int. Test Conf.*, 1997, pp. 406–413.
13. Dekker, R. et al., A Realistic Self-Test Machine for Static Random Access Memories, *Proc. Int. Test Conf.*, 1988, pp. 353–361.
14. Franklin, M., and K. K. Saluja, Built-in Self-Testing of Random-Access Memories, IEEE Computer, Vol. 23, No. 10, October, 1990, pp. 45–56.
15. Ohsawa, T. et al., A 60-ns 4-Mbit CMOS DRAM With Built-In Self-Test Function, *IEEE J. Solid-State Circuits*, Vol. 22, No. 5, October 1987, pp. 663–668.
16. Sridhar, T., A New Parallel Test Approach for Large Memories, *IEEE Des. Test*, Vol. 3, No. 4, August 1986, pp. 15–22.
17. Altnether, J. P., and R. W. Stensland, Testing Redundant Memories, *Electron. Test*, Vol. 6, No. 5, May 1983, pp. 66–76.
18. Shannon, C. E., The Mathematical Theory of Communication, *Bell Syst. Tech. J.*, Vol. 27, July and October, 1948.
19. May, T. C., and M. H. Woods, Alpha-Particle-Induces Soft Errors in Dynamic Memories, *IEEE Trans. Electron. Dev.*, ED-26, No. 1, January 1979, pp. 2–9.
20. Bossen, D. C., and M. Y. Hsiao, A System Solution to the Memory Soft Error Problem, *IBM J. Res. Dev.*, Vol. 24, No. 3, May 1980, pp. 390–398.
21. Khan, A., Fast RAM Corrects Errors on Chip, *Electronics*, September 8, 1983, pp. 126–130.

22. Levine, L., and W. Meyers, Semiconductor Memory Reliability with Error Detecting and Correcting Codes, *Computer*, Vol. 9, No. 10, October 1976, pp. 43–50.
23. Palfi, T. L., MOS Memory System Reliability, *IEEE Semiconductor Test Symp.*, 1975.
24. Travis, B., IC's and Semiconductors, *EDN*, December 17, 1982, pp. 40–46.
25. Wolfe, C. F., Bit Slice Processors Come to Mainframe Design, *Electronics*, February 28, 1980, pp. 118–123.
26. Peterson, W. W., *Error Correcting Codes*, Chapter 5, M.I.T. Press, Cambridge, MA., 1961.
27. Koike, H. et al., A BIST Scheme Using Microprogram ROM For Large Capacity Memories, *Proc. Int. Test Conf.*, 1990, pp. 815–822.
28. Nicolaidis, M., Transparent BIST For RAMs, *Proc. Int. Test Conf.*, 1992, pp. 598–607.
29. Prince, Betty, *Semiconductor Memories: A Handbook of Design, Manufacture, and Application*, 2nd ed., John Wiley & Sons, New York, 1991 (reprinted, 1996).
30. van de Goor, A. J., *Testing Semiconductor Memories: Theory and Practice*, Wiley & Sons, New York, 1991 (reprinted, 1996).

CHAPTER 11

I_{DDQ}

11.1 INTRODUCTION

Test strategies described in previous chapters relied on two concepts: controllability and observability (C/O). Good controllability makes it easier to drive a circuit into a desired state, thus making it easier to sensitize a targeted fault. Good observability makes it easier to monitor the effects of a fault. Solutions for solving C/O problems include scan path and various ad-hoc methods. Scan path reduces C/O to a combinational logic problem which, as explained in Chapter 4, is a solved problem (theoretically, at least).

I_{DDQ} monitoring is another approach that provides complete observability. Current drain in a properly functioning, fully static CMOS IC is negligible when the clock is inactive. However, when the IC is defective, due to the presence of leakage in the circuit, or possibly even to an open, current flow usually becomes excessive. This rise in current flow can be detected by monitoring the current supplied by the tester. How effective is this technique for spotting defective ICs? In one study, it was shown that I_{DDQ} testing with a test program that provided 60% coverage of stuck-at faults provided the same AQL as a test program with 90% stuck-at coverage without I_{DDQ}.[1]

The stuck-at fault model that we have been dealing with up to this point is not intended to address qualitative issues; its primary target is solid defects manifested as signals stuck-at logic 1 or logic 0. An IC may run perfectly well on a tester operating at 1 or 2 MHz, at room temperature, but fail in the system. Worse still, an IC may fail shortly after the product is delivered to the customer. This is often due to leakage paths that degrade to catastrophic failure mode shortly after the product is put into service.

11.2 BACKGROUND

The CMOS circuit was patented in 1963 by Frank Wanlass.[2] His two-transistor inverter consumed just a few nanowatts of standby power, whereas equivalent bipolar circuits of the time consumed milliwatts of power in standby mode. During

Digital Logic Testing and Simulation, *Second Edition*, by Alexander Miczo
ISBN 0-471-43995-9 Copyright © 2003 John Wiley & Sons, Inc.

the 1970s, companies began measuring leakage of CMOS parts to identify those that had excessive power consumption.[3] At times it was a useful adjunct to the traditional functional testing for stuck-at faults, and at other times it was critical to achieve quality levels required by customers.

The classic stuck-at fault model, while identifying unique signal paths (cf. Section 7.5) and providing a means for quantitatively measuring the completeness of a test for these paths, does not model many of the fault classes that can occur, particularly in deep submicron circuits. In fact, as was pointed out in Section 3.4 that the stuck-at fault can be thought of as a behavioral model for very low level behavioral devices, namely, the logic gates.

Faults such as high-resistance bridging shorts, inside a logic gate or between connections to adjacent gates, may not be visible during a functional test. A leakage path may cause path delay, so the circuit does not operate correctly at speed, but it may operate correctly if the circuit is tested at a speed much slower than its design speed, since there may be enough time for a charge to build up and force the gate to switch. Shorts between signal runs on the die are usually overlooked during functional testing, because, in general, there is no fault model to determine if they have been tested. If there were fault models for these shorts, perhaps generated by a layout program, the number of these faults would be prohibitively large and would aggravate a frequently untenable fault simulation problem (cf. Section 3.4).

Excess current detected during test may indicate reliability problems. The inverter depicted in Figure 11.1 has a short circuit from gate to drain of Q_1. In normal operation, when input A switches from 0 to 1, there is a brief rush of current between V_{DD} and ground. Shortly thereafter, a high at the gate of Q_1 causes a near complete cutoff of current, the measured flow typically being a few nanoamperes. This minuscule current flow is quite important in battery operated applications, ranging from human implants to laptop computers. However, because of the defect, there is a path from ground, through the drain of Q_2, to the source of Q_1 and then to the gate. The output F in this example will likely respond with the correct value, since it is logically connected to ground through Q_2, but current flow will be excessive, and there is the possibility of a catastrophic failure in the future.

Interestingly, although much attention is given to detection of shorts by I_{DDQ}, it can also detect open circuits. When an open occurs, it is often the case that neither

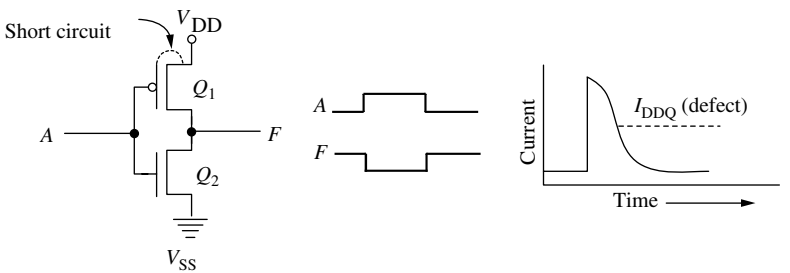


Figure 11.1 CMOS inverter.

transistor of a transistor pair is completely turned off. As a result, a leakage path from ground to V_{DD} exists. This is significant because, in conventional stuck-fault testing, a two-vector combination is required to detect stuck-open faults in CMOS circuits (cf. Section 7.6.2).

11.3 SELECTING VECTORS

In order to measure leakage current, the circuit must be in a fully initialized state. I_{DDQ} measurements must be made on quiet vectors—that is, vectors with very little leakage current. During simulation, those vectors for which indeterminate values are detected must immediately be eliminated as candidates for current measurement. During test, when the circuit reaches a vector at which a current measurement is to be made, the circuit must be held in a steady state for a sufficient duration to allow all switching transients to subside. Some design rules include:

No pullups or pulldowns.

No floating nodes.

No logic contention.

If analog circuits appear in the design, they should be on separate power supplies.

No unconnected inputs on unused logic.

The purpose of these design rules is to prevent excess current flows during quiescent periods. Pullups and pulldowns provide resistive paths to ground or power. On average, a node is going to be at logic 0 half the time and at logic 1 half the time. If the node is at logic 0 and is connected to a pullup, a path exists for current flow. Floating inputs may stabilize at a voltage level somewhere between ground and V_{DD}, thus providing a current path. Incompletely specified busses can be troublesome. For example, if a bus has three drivers, a logic designer may design the circuit in such a way that the select logic floats the bus when no driver is active. Hence, any inputs driven by the bus will be floating. Bus keeper cells are recommended to prevent floating buses.[1]

In general, any circuit configuration that causes a steady current drain from the power supply runs the risk of masking failure effects, since the effectiveness of I_{DDQ} relies on the ability to distinguish between the very low quiescent current drain for a defect-free circuit and the high current caused by a defect. Interestingly, redundant logic, which is troublesome for functional testing, does not adversely affect I_{DDQ} testing. In fact, I_{DDQ} can detect defects in redundant logic that a functional test cannot detect.

11.3.1 Toggle Count

Toggle count has been used for many years as a metric for evaluating the thoroughness of gate-level simulations for design verification. When schematic entry was the primary medium for developing logic circuits, and the level of abstraction was logic

gates, toggle count could be used to identify nodes on the schematics that were never toggled to a particular value. Those nodes were then targeted during simulation, the objective being to get all or nearly all nodes toggled to both 1 and 0.

Since one of the objectives of I_{DDQ} is to identify circuits with short circuits between signal lines and power or ground, the toggle count can be an effective method for determining the effectiveness of a given test. If a particular set of test vectors has a high toggle percentage, meaning that a high percentage of nodes toggled to both 1 and 0, then it is reasonable to expect that a high percentage of shorts will be detected.

The computation is quite straightforward: simply identify the gate that is driving each line in the circuit and note whether it has toggled to a 1 or 0 at the end of each vector. Then, during simulation, the first step is to determine whether or not the vector can be used for I_{DDQ}. Recall that a vector cannot be a candidate if the circuit is not yet fully initialized, or if there is bus contention. If the vector is a candidate, then determine how many previously untoggled nodes are toggled by this vector. Since there is usually a limit on the number of vectors for which the tester can make I_{DDQ} measurements, it is desirable to select vectors such that each vector selected contributes as many new nodes as possible to the collection of toggled nodes.

The first vector that meets acceptance criteria is generally going to provide about 50% coverage, since every node is at 1 or 0. A scheme described in the Quietest method (next section), but that is also applicable here, establishes a percentage of the untoggled node values as an objective. As an example, an objective might be established that bars a vector from being selected unless it toggles at least 10% of the currently untoggled node values. As toggle coverage increases, the 10% selection criteria remains, but the absolute number of newly toggled node values decreases.

This procedure can be applied iteratively. For example, a given percentage may be too restrictive; as a result, no new vectors are selected after some toggle coverage is reached. Those vectors can be retained, and then simulation can be rerun with a lower percentage threshold, say 5%. This will usually cause additional vectors to be selected. If the maximum allowable number of vectors has not been reached, and the toggle coverage has not yet reached an acceptable level, this procedure can again be repeated with yet another lower selection percentage.

11.3.2 The Quietest Method

The *quietest method* is based on the observation that six shorts can occur in a single MOS transistor:[4]

f_{GS} gate and source
f_{GD} gate and drain
f_{SD} source and drain
f_{BS} bulk and source
f_{BD} bulk and drain
f_{BD} bulk and gate

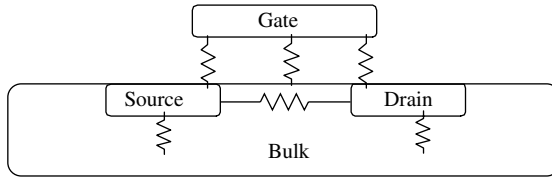
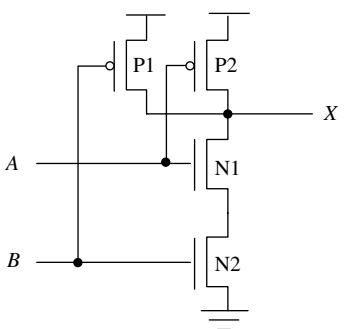


Figure 11.2 MOS transistor short fault model.

These shorts are seen in Figure 11.2. The approach used in this method is applicable at the transistor level or at the macrocell level. It begins with a table for a particular cell, which could be a simple logic gate, or a full-adder, or a considerably more complex circuit. All input combinations to the cell are fault-simulated at the transistor level. This list of transistor shorts permits I_{DDQ} fault simulation of the entire circuit to be accomplished hierarchically.

The first step is to simulate each transistor or macrocell and to fault-simulate each of the faults. A table is created for each cell, listing I/O combinations versus faults detected (see Figure 11.3). The NAND gate, Figure 11.3(a), is simulated, and the table of Figure 11.3(b) is constructed. This table is a matrix of dimension $m \times n$, where $m = 2^k$ is the number of rows, and k is the number of I/O pins. The circuit shown in Figure 11.3 has two inputs and one output, so there are 2^3 rows.

The number of columns, n, corresponds to the number of transistors. Each entry in the table is a two-character octal number. The six bits corresponds to the six transistor faults, as defined in Figure 11.3(c). The all-zero row entries for combinational logic correspond to combinations that cannot occur. For example, row 2 corresponds



(a)

i	N1	N2	P1	P2
0	0	0	0	0
1	22	0	43	43
2	0	0	0	0
3	26	43	43	0
4	0	0	0	0
5	70	26	0	43
6	43	43	26	26
7	0	0	0	0

(b)

f_{BG}	f_{BD}	f_{BS}	f_{SD}	f_{GD}	f_{GS}

(c)

Figure 11.3 Lookup table for I_{DDQ} faults.

to the combination *A, B, X* = (0,1,0), which is inconsistent with the definition of a NAND gate. Note, however, that some combinations in sequential circuits may rely on the presence of feedback.

Once the table is created, it can be used to compute I_{DDQ} coverage for the cell during normal logic simulation. At the end of each vector, the input combination on each macrocell is examined. If the combination has not been generated by any previously selected I_{DDQ} vector, then any short faults detected by this combination, and not previously marked as detected, can be selected and tallied for the current vector. After all cells have been examined, the incremental improvement in fault coverage for the vector can be computed. If the vector satisfies some criteria, such as that described in the previous subsection, it can be accepted and added to the collection of vectors for which I_{DDQ} measurements are to be made.

11.4 CHOOSING A THRESHOLD

One of the problems associated with I_{DDQ} is choice of a current threshold. Different devices exhibit different amounts of leakage current. Even different devices of the same die size may have significantly different amounts of leakage current, depending on the kind of logic and/or memory that is contained on the die. Furthermore, the same device, when tested at wafer sort and at package test, will exhibit different leakage. The target application of the IC will influence the leakage threshold: Manufacturers of ICs for portable applications or human implants will have much more stringent requirements on leakage current.

The issue is further complicated by the fact that different vectors from the same test vector set can have noticeably different leakage currents. As a result, it is a nontrivial task to establish a threshold for current. A threshold that is too lax results in keeping devices that should be discarded. Conversely, a threshold that is too rigorous results in discarding good devices. One source suggests that if I_{DDQ} of the device under test is greater than 100 μA for all vectors under normal conditions, the IC cannot be tested by means of I_{DDQ} measurement.[5]

Determining a threshold starts with a histogram of I_{DDQ} current versus number of devices that occur in each bin of the histogram. Figure 11.4 shows a histogram for 11,405 microcontrollers.[6] The author uses I_{SSQ} to denote the fact that current is measured at V_{SS} rather than V_{DD}. In an IEEE QTAG (Quality Test Action Group) survey, respondents were asked where they would set a threshold for the data in Figure 11.4.[7] The following results were obtained:

500–100 μA	3
100–50 μA	7
50–25 μA	4
25–10 μA	3
10–5 μA	6
<5 μA	5

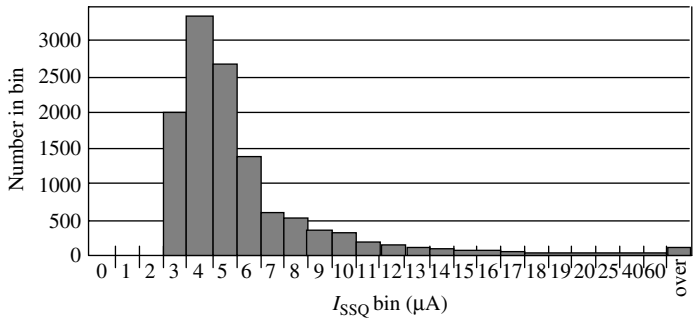


Figure 11.4 Distribution of I_{SSQ}.

One experiment that was conducted attempted to correlate I_{DDQ} results with the results of functional tests. In this experiment, I_{DDQ} was measured in die that passed functional test with high stuck-fault coverage and in die that failed the same functional tests. It was shown that 96% of parts passing the functional test measured less than 1 μA, while only 2% of parts reading greater than 1 mA passed functional test.[1] Conversely, of parts failing functional test, 83% gave I_{DDQ} readings of over 1 mA, while only 15% read less than 1 μA.

It has been recommended that I_{DDQ} measurements be made at the highest possible V_{DD} in order to ensure detection of defects that have strong nonlinear characteristics.[8] The authors of this study report that a defective IC leaked 10 nA at 5 V but 29.3 μA at 6.2 V. These same authors point out that a design that was amenable to I_{DDQ} testing had, nonetheless, some particular vectors in which I_{DDQ} values were on the order of 265 μA. In general, it seems safe to say that the selection of a threshold will, of necessity, be empirical, since there is no hard and fast rule. Measurements such as those described here, involving measurement of I_{DDQ} for those that pass versus those that fail functional test, help to shed light on the subject. Measurement of I_{DDQ} from lots with different yields, along with die from different points on the wafer and at different voltages and after different periods of quiescence, can help to influence one's judgment as to where to set the threshold.

11.5 MEASURING CURRENT

A proposed circuit for measuring I_{DDQ} current flow that has come to be known as the Keating–Meyer circuit is shown in Figure 11.5.[8] At the beginning of the period, Q_1 is on and provides a short circuit between C_1 and C_2, maintaining full voltage to the DUT. Eventually, Q_1 is turned off and static current to the DUT is obtained exclusively from C_1. The value of C_1 is determined from the relationship $C_1 = I \cdot t/V$, where I is the desired measurement resolution, t is the elapsed time within which it is desired to make a measurement, and V is the voltage resolution at the op amp.

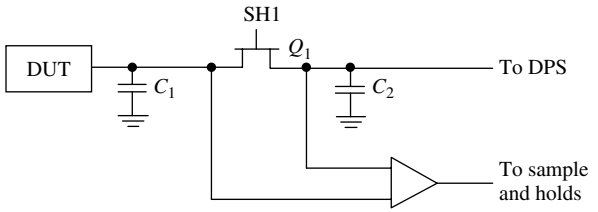


Figure 11.5 I_{DDQ} pass/fail circuit.

Example Suppose we want a measurement resolution of 25 μA within 500 ns, along with 10 mV at the op amp:

$$C_1 = \frac{It}{V} = \frac{25\ \mu A \cdot 500\ ns}{10\ mV} = 1250\ pF$$

For the capacitance value of 1250 pF, if we wish to limit voltage drop at the DUT to 1.0 V ($V_{CC} > 4$ V), for a defect-free device ($I_{DD} < 25$ μA), then the voltage drop across Q_1 must be measured within $t_1 < CV/I = 50$ μS. ■■

The circuit in Figure 11.5 can also be used to measure switching currents, as well as static I_{DD}. For example, if a 1.0-A peak current is assumed, lasting 5 ns, then for a desired resolution of 100 μA at 10 mV and for a 500-ns I_{DD} measurement time, $C_1 = 100\ \mu A * 500\ ns/10\ mV = 5000$ pF.

Turn off Q_1 and clock the device at $t = 0$ ns. Then sample the drop across Q_1 at $t = 100$ ns. The total charge delivered by C_1 is

$$Q_1 = \int i\ dt = (1\ A \cdot 5\ ns) + (100\ \mu A \cdot 97\ ns) = 5\ nC + 9.7\ pC = 5\ nC$$

The voltage across Q_1 equals $V = Q/C = 5$ nC/.005 μF = 1 V. In these equations, the value of C_1 is critical. An optimal value must be selected in order to avoid unnecessarily increasing test time or producing excessive V_{CC} drop at the DUT.

The QuiC-Mon circuit builds on the Keating–Meyer concept.[9] Figure 11.6 illustrates the QuiC-Mon circuit. The key difference is that QuiC-Mon takes the time derivative of the voltage at V_{DD}. As a result, the constant-slope waveform is converted into a step function and settling time improves significantly, allowing faster measurement rates. Measurements with QuiC-Mon can be taken using I_{DDQ} or I_{SSQ}. However, I_{SSQ} provides more accurate measurements at input pins with internal pullups when the pin is driven low. The transfer function for the QuiC-Mon circuit of Figure 11.6 is

$$V_S = \frac{R_3}{R_2} V_1 = \frac{R_3}{R_2} R_1 C_1 \frac{dV_{SS}}{dt} = \frac{R_3}{R_2} R_1 \frac{C_1}{C_1 + C_{SS}} I_{SSQ}$$

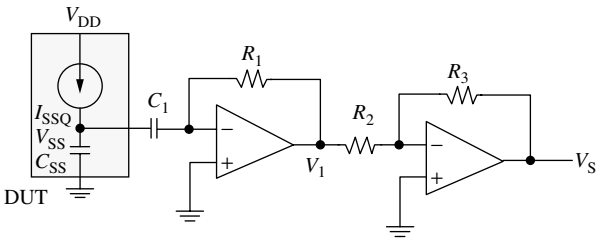


Figure 11.6 The QuiC-Mon circuit.

If capacitor C_1 is large compared to the DUT capacitance C_{SS}, the transfer function is

$$V_S = R_1 \frac{R_3}{R_2} I_{SSQ}$$

When using the monitor, a number of factors must be taken into consideration in order to achieve accuracy and speed. It is important to minimize the physical length of the V_{SS} path between the DUT and QuiC-Mon to reduce noise and inductance. It is recommended that the monitor be within 2 or 3 cm of the DUT. For I_{DDQ} testing bypass capacitance should be minimized so measurement speed is unaffected. For I_{SSQ} testing, bypass capacitance is not a significant issue.

The resistor R1 can be increased to amplify QuiC-Mon's output. However, after a point, larger values require low-pass filtering. The circuit can achieve gains of up to 500 mV/μA at 250 kHz, which is sufficient for high-speed, submicroampere resolution. In some applications, transient settling time limited measurement speeds to 100 kHz.

11.6 I_{DDQ} VERSUS BURN-IN

Burn-in is a process of continuously energizing a circuit, usually under extreme voltage or environmental conditions, in order to precipitate failures of devices that are marginal performers due to fabrication imperfections. It is well known that most devices that fail will do so within a few days or weeks of their initial purchase. This is illustrated in Figure 11.7. Some devices will pass the initial testing phase, when the testing is performed at nominal values of the key parameters, but will fail shortly after when put into operation. By elevating parameters such as voltage and temperature, many of the devices susceptible to early life failures can be identified and discarded before they are packaged and shipped to customers.

There is growing evidence that an effective I_{DDQ} program can serve the same purpose as a burn-in program. One of the more prevalent failures common to CMOS circuits is the gate-oxide short (GOS). The GOS may create a high-resistance leakage current path that does not initially affect performance because of the high noise

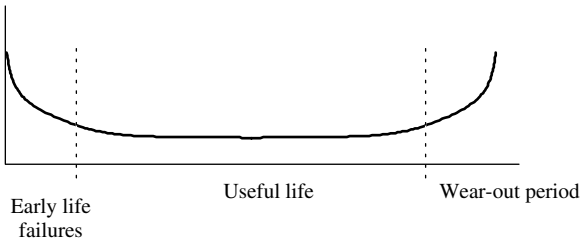


Figure 11.7 Bathtub curve.

margin of the field-effect transistor (FET). Eventually, over time, the resistance decreases and the device fails.

In a paper previously cited in this chapter,[1] the author evaluated the effects of I_{DDQ} on burn-in. It was found that the use of I_{DDQ} reduced burn-in failures by 80%, whereas adding additional functional tests had only a marginal effect on reducing burn-in failures. Another study was performed on ASICs returned from the field. The author found that nearly 70% of the parts would have failed an I_{DDQ} test.[10] In another study the author subjected parts failing an I_{DDQ} test to a 1000-hour life test. The experiment revealed that about 8% of the parts that failed the I_{DDQ} test failed the 1000-hour life test. Yet another study was conducted on 2100 die that failed I_{DDQ}. When subjected to burn-in, the failure rate was 10 times greater than that of a control sample.[11] In yet one more previously cited study, the number of parts failing a 24-hour burn-in was reduced from a failure rate of 448 ppm to a rate of 25.6 ppm.[6]

A study performed at Intel was used to justify the use of I_{DDQ} as a major part of the test strategy on the i960JX CPU.[12] The goal was to achieve ZOBI (zero hour burn-in). This decision was shown to save about 1.25 million dollars as a result of reduced capital costs, reduced test costs per part, and yield improvement. In order to achieve ZOBI, it was necessary to demonstrate a defects per million (DPM) of less than 1000 (0.1% DPM). It was also necessary to have at least 30% of burn-in hardware in place for contingencies, and it was necessary to have SBLs (statistical bin limits) on key bins at wafer sort.

The tool used by this division of Intel was an I_{DDQ} fault simulator called iLEAK. It generated tables based on the Quietest method (Section 11.3.2). The use of toggle coverage and an option in iLEAK called *fastileak* helped to reduce the amount of computation by screening the vector sets and choosing candidate vectors for iLEAK to evaluate. At the end of that process, seven vectors were chosen. This set of vectors was augmented with another six vectors, bringing the total number to 13.

The i960 CPU is a two-phase clock design, and only one of the phases is static, so it was necessary to change the vector format to ensure that the clock would stop during the static phase. Through experimentation it was determined that the delay time needed to measure leakage current was 20 ms per I_{DDQ} strobe.

A key concern in setting up the I_{DDQ} process was to ensure defect detection without overkill—that is, discarding excessive numbers of good die. To achieve this, it

was determined that the test limit would be statistically based. This would be accomplished by gathering I_{DDQ} values from functionally good die from several wafers across a skew lot, one in which material has been intentionally targeted to reside within parametric corners of the wafer fabrication process. Skew parameters being used were gate oxide and poly critical dimensions. From the skew lot, I_{DDQ} values were gathered and displayed in the form of cumulative plots. Outliers, samples that had current well outside the range of the other good die, were removed. The remaining samples represented a Gaussian population from which a mean and standard deviation could be generated. Limits for each vector were determined using mean plus 4σ.

It was recognized that using I_{DDQ} at wafer sort provided the biggest payback, because identifying and discarding ICs with high leakage current at wafer sort eliminated the expense of packaging and testing the packaged parts. However, it was not known whether the methodology used for wafer sort would also be needed at package test to satisfy the DPM requirements for elimination of burn-in. A factor that had to be considered was the temperature at test. Wafer sort was performed at lower temperatures, and I_{DDQ} provided better defect detection at lower temperatures.

The investigation of the efficacy of I_{DDQ} at package test was designed to determine whether or not it was needed in order to eliminate burn-in. An experiment was designed to determine the I_{DDQ} limit and measure its effectiveness. The first goal was to determine if I_{DDQ} was needed at package test. A second goal, assuming that I_{DDQ} was necessary, was to determine limits that would minimize yield losses while screening out latent failures.

Because the test temperature at package test was higher, thus increasing transistor subthreshold leakage, units were tested using the following test flow:

- Use wafer sort I_{DDQ} limits.
- Measure I_{DDQ} (but without a pass/fail condition).
- Test units with new I_{DDQ} limits.

It was quickly learned that the sort I_{DDQ} limits could not be used at package test. The I_{DDQ} values at package test had high lot-to-lot variability and strobe-to-strobe variability. The 13 vectors used to measure I_{DDQ} were divided into two categories, high strobes and low strobes, based on the leakage current that was measured. Six of the strobes fell into the high strobes category, while seven fell into the low strobes category. For the low strobes a limit of 53 μA was set, while for the high strobes the limit was set at 3 mA. These limits would produce about 3% I_{DDQ} fallout at package test.

The next step in the evaluation of package test I_{DDQ} was to run some ICs through a test sequence to determine whether or not package test I_{DDQ} actually detects failing ICs. Devices that failed a high-temperature I_{DDQ} were measured to get I_{DDQ} values before burn-in. After burn-in, a *post burn-in check (PBIC)* revealed that all the devices that were I_{DDQ} failures before burn-in passed all functional testing after burn-in. From this it was concluded that I_{DDQ} testing at package test did not provide any additional quality or reliability.

The ZOBI evaluation was the next step in the process. This involved several phases, during which the goal was to achieve <0.1% PBIC fallout. After some initial steps in which testing was performed using a test program with about 82% fault coverage, the fallout percentage was 0.2%. Then I_{DDQ} *screen at sort* and package test were introduced. The screen at sort reduced the PBIC fallout to 0.07%. The screen at package test was shown to be unnecessary. ZOBI was achieved, and a system was put in place to monitor the device based on sampling and *statistical bin limits (SBL)*. If I_{DDQ} at sort is found to be excessively high for a particular lot, the SBLs trigger the lot for burn-in.

11.7 PROBLEMS WITH LARGE CIRCUITS

An investigation of opens in CMOS circuits showed that circuits could function correctly at low frequencies, in the presence of narrow opens occurring as a result of electromigration.[13] One of the possibilities that was considered was capacitive coupling with signal lines running adjacent to the open-circuited line. It was determined that this was not the reason for the circuit responding correctly. Further investigation suggested that tunneling current was the mechanism by which electrons managed to pass from one side of the open to the other side. However, this property only occurred at low frequencies. At higher frequencies there was insufficient time to charge and discharge the transistor gates. An implication of this is that a delay fault model should be used to detect very small opens caused by electromigration.

The changing threshold voltage V_t represents a more serious problem for I_{DDQ} as IC feature sizes shrink into the deep submicron region.[14] I_{DDQ} requires that defective devices have a significantly greater quiescent current than the good devices. This difference between faulty and good circuits must take into account variations in quiescent current for the good circuit attributable to process variations in the good circuit as well as variations in the measurement accuracy of the tester. However, as device geometries shrink and transistor count increases, quiescent current grows for good devices. This leads to a collapsing of the range of quiescent currents for good and faulty devices.

The results of scaling are depicted in Figure 11.8. Dimensions of the scaled device are shrunk by a factor S. Since dimensions are shrunk in two dimensions, die size can be reduced by approximately S^2. The shorter distances that signals have to

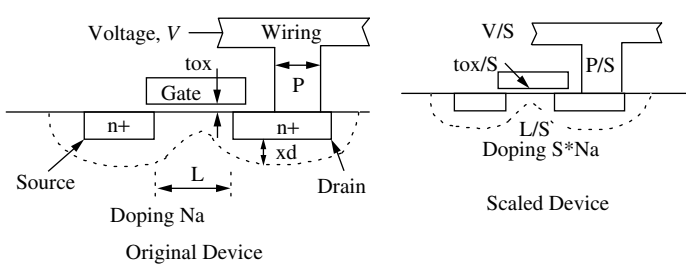


Figure 11.8 Device before and after scaling.

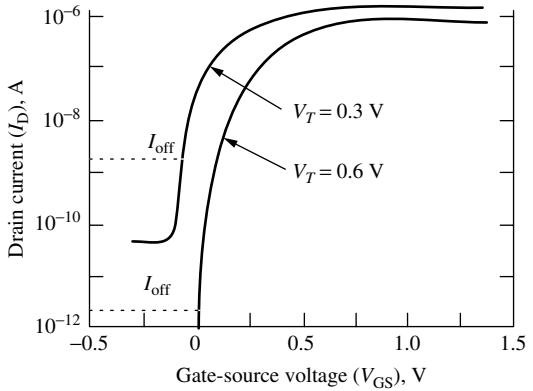


Figure 11.9 Off-state leakage current for two values of V_T.

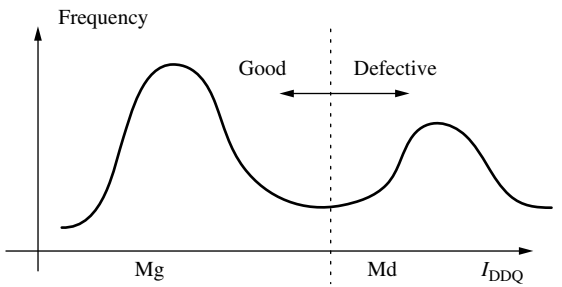


Figure 11.10 General distribution of I_{DDQ} values seen in CMOS networks.

travel, along with reduced capacitance, contribute to improved speed. However, the shorter channel requires a lower V_T value. If voltages are held fixed while dimensions are reduced, internal electric fields will rise, increasing the reliability risk, particularly at the insulation between the gate and the substrate where tox (oxide thickness) has been reduced. Consequently, the supply voltage V_{DD} must be reduced in order to minimize the electric fields in the transistors, as well as to reduce power consumption.

In order to provide optimum switching speed of transistors, the threshold voltage V_T is normally about 15–20% of V_{DD}. Thus, as V_{DD} continues to drop, eventually dropping to 1.8 V, V_T will be on the order of 0.3 V. A result of this, shown in Figure 11.9, is that I_{off}, and therefore I_{DDQ}, rises by three orders of magnitude.[15] A result of this is a shift in the distribution of I_{DDQ} values for Mg, the good circuit, and Md, the defective circuit, as shown in Figure 11.10. Mg shifts to the right and Md shifts to the left, making it more difficult to distinguish between good and defective devices. The following table[14] illustrates the trends in CMOS technology with scaling V_t proportional to V_{DD} at 25°C.

Year	Gates ($\times 10^6$)	V_{DD} (V)	L_{eff} (μm)	Max i_{off} (nA/μm)
1995	5	3.3–2.5	0.25–0.45	5–10
1998	14	2.5–1.8	0.21–0.25	10–20
2001	26	1.8–1.5	0.1–0.15	20
2004	50	1.5–1.2	0.1–0.15	30
2007	210	<1.2	<0.1	30

The implication of the trends predicted for CMOS circuits is that I_{DDQ}, at least for the larger circuits, will be less effective and, in fact, it may be impossible to exploit I_{DDQ} for the largest circuits, where its contribution to distinguishing between faulty and fault-free circuits is most critically needed. Some design-for-test suggestions have been proposed to address this problem. For those circuits where the problem is most severe, and the payback most pronounced, the suggestions may be reasonable.

11.8 SUMMARY

A number of methods for creating and evaluating test programs have been examined up to this point. In all cases, the essential goal has been to control and observe the circuit. Many quite elaborate methods and circuits have been designed to enhance either controllability or observability, or both. In the case of CMOS circuits, the observability provided by I_{DDQ} measurements is virtually free. Rather modest requirements in the way of design rules are sometimes necessary. But the payoff, when these rules are enforced, can be significant when weighed against the cost. Expressed another way, adhering to the rather modest I_{DDQ} design rules provides tremendous leverage. Exploiting this free observability simply involves monitoring power supply current.

A quite straightforward technique for selecting vectors that provide high I_{DDQ} coverage involves the use of CAD tools that compute toggle coverage. This is a design verification technique that has been in use for three decades, and numerous tools exist that are capable of performing this task. Code coverage tools for evaluating RTL designs offer a potential for selecting test vectors before the design is even converted to its gate-level equivalent.

There is some concern for the future of I_{DDQ} as circuits grow larger. Bear in mind that applications range from rather small controllers to large, multi-million transistor, state-of-the-art microprocessors. For the smaller applications, ranging from a few thousand gates up to a couple of hundred thousand gates, and where aggressive shrinkage of feature size is not a priority, I_{DDQ} is in use and has been demonstrated to be a viable test solution. For larger applications, where shrinking feature size and growing numbers of transistors poses a threat to the applicability of I_{DDQ}, some DFT concepts

specific to I_{DDQ} will need to evolve. Since these are the most design-intensive and are likely to involve large design teams, including a cadre of design-for-test experts, solutions that exploit I_{DDQ} may continue to find their way into the designs.

PROBLEMS

11.1 For the NAND gate in Figure 11.3, identify all faults detected when the input combination A, B = (1, 0) appears at the inputs. What is the total fault coverage for that input combination?

11.2 In Figure 3.3 there is an AND-OR-Invert circuit. Create a table of I/O values versus faults. What is the fault coverage when the sequence X_1, X_2, Y_1, Y_2 = {(0010, 0110, 1000)} is applied to that circuit.

11.3 Discuss the relative merits of the two methods described for computing I_{DDQ} coverage—that is, the Quietest method versus measuring toggle count.

REFERENCES

1. Wiscombe, Paul C., A Comparison of Stuck-at Fault Coverage and I_{DDQ} Testing on Defect Levels, *Proc. IEEE Int. Test Conf.*, 1993, pp. 293–299.
2. Riezenman, Michael J., Wanlass's CMOS Circuit, *IEEE Spectrum*, May 1991, p. 44.
3. Baker, K., A. Bratt, A. Richardson, and A. Welbers, Development of a Class 1 QTAG Monitor, *Proc. IEEE Int. Test Conf.*, 1994, pp. 213–222.
4. Mao, Weiwei, R. K. Gulati, D. K. Goel, and M. D. Ciletti, QUIETEST: A Quiescent Current Testing Methodology for Detecting Leakage Faults, *Proc. Int. Conf. Comput. Aided Des.*, 1990, pp. 280–283.
5. Application Note 398-3, Measuring CMOS Quiescent Power Supply Current (I_{DDQ}) with the HP 82000, Hewlett Packard, 1993.
6. Wallquist, K. M., On the Effect of I_{SSQ} Testing in Reducing Early Failure Rate, *Proc. IEEE Int. Test Conf.*, 1995, pp. 910–915.
7. Quality Test Action Group, QTAG Main Meeting: Minutes and Assoc. Material, *Proc. Int. Test Conf.*, 1994.
8. Keating, M., and D. Meyer, A New Approach to Dynamic IDD Testing, *Proc. IEEE Int. Test Conf.*, 1987, pp. 316–321.
9. Wallquist, K. M., Achieving I_{DDQ}/I_{SSQ} Production Testing with QuiC-Mon, *IEEE Des. Test Comput.*, Fall 1995, pp. 62–69.
10. McEuen, Steven D., I_{DDQ} Benefits, Proc. *IEEE VLSI Test Conf.*, p. 285, 1991.
11. Kane, J., *Proc. VLSI Test Symp.*, 1994.
12. Henry, T. R., and Thomas Soo, Burn-in Elimination of a High Volume Microprocessor Using I_{DDQ}, *Proc. IEEE Int. Test Conf.*, 1996, pp. 242–249.

13. Henderson, C. L., J. M. Soden, and C. F. Hawkins, The Behavior and Testing Implications of CMOS IC Logic Gate Open Circuits, *Proc. IEEE Int. Test Conf.*, 1991, pp. 302–310.

14. Williams, T. W. et al., I_{DDQ} Test: Sensitivity Analysis of Scaling, *Proc. IEEE Int. Test Conf.*, 1996, pp. 786–792.

15. Soden, J. M., C. F. Hawkins, A. C. Miller, Identifying Defects In Deep-Submicron CMOS ICs, *IEEE Spectrum*, September 1996, pp. 66–71.

CHAPTER 12

Behavioral Test and Verification

12.1 INTRODUCTION

The first 11 chapters of this text focused on manufacturing test. Its purpose is to answer the question, "Was the IC fabricated correctly?" In this, the final chapter, the emphasis shifts to design verification, which attempts to answer the question, "Was the IC designed correctly?" For many years, manufacturing test development and design verification followed parallel paths. Designs were entered via schematics, and then stimuli were created and applied to the design. Design correctness was confirmed manually; the designer applied stimuli and examined simulation response to determine if the circuit responded correctly. Manufacturing correctness was determined by simulating vectors against a netlist that was assumed to be correct. These vectors were applied to the fabricated circuit, and response of the ICs was compared to response predicted by the simulator. Thoroughness of design verification test suites could be evaluated by means of toggle counts, while thoroughness of manufacturing test suites was evaluated by means of fault simulation.

In recent years, most design starts have grown so large that it is not feasible to use functional vectors for manufacturing test, even if they provide high-fault coverage, because it usually takes so many vectors to test all the functional corners of the design that the cost of the time spent on the tester becomes prohibitive. DFT techniques are needed both to achieve acceptable fault coverage and to reduce the amount of time spent on the tester. A manufacturing test based on scan targets defects more directly in the structure of the circuit. A downside to this was pointed out in Section 7.2; that is, some defects may best be detected using stimuli that target functionality.

While manufacturing test relies increasingly on DFT to achieve high-fault coverage, design verification is also changing. Larger, more complex designs created by large teams of designers incorporate more functionality, along with the necessary handshaking protocols, that must be verified. Additionally, the use of core modules, and the need to verify equivalence of different levels of abstraction for a given design, have made it a greater challenge to select the best methodology for a given

Digital Logic Testing and Simulation, *Second Edition*, by Alexander Miczo
ISBN 0-471-43995-9 Copyright © 2003 John Wiley & Sons, Inc.

design. What verification method (or methods) should be selected? Tools have been developed to assist in all phases of support for the traditional approach—that is, apply stimuli and evaluate response. But, there is also a gradual shift in the direction of formal verification.

Despite the shift in emphasis, there remains considerable overlap in the tools and algorithms for design verification and manufacturing test, and we will occasionally refer back to the first 11 chapters. Additionally, we will see that, in the final analysis, manufacturing test and design verification share a common goal: reliable delivery of computation, control, and communication. If it doesn't work correctly, the customer doesn't care whether the problem occurred in the design or the fabrication.

12.2 DESIGN VERIFICATION: AN OVERVIEW

The purpose of design verification is to demonstrate that a design was implemented correctly. By way of contrast, the purpose of design validation is to show that the design satisfies a given set or requirements.[1] A succinct and informal way to differentiate between them is by noting that[2]

> Validation asks "Am I building the right product?"
>
> Verification asks "Am I building the product right?"

Seen from this perspective, validation implies an intimate knowledge of the problem that the IC is designed to solve. An IC created to solve a problem is described by a data sheet composed of text and waveforms. The text verbally describes IC behavior in response to stimuli applied to its I/O pins. Sometimes that behavior will be very complex, spanning many vectors, as when stimuli are first applied in order to configure one or more internal control registers. Then, behavior depends on both the contents of the control registers and the applied stimuli. The waveforms provide a detailed visual description of stimulus and response, together with timing, that shows the relative order in which signals are applied and outputs respond.

Design verification, on the other hand, must show that the design, expressed at the RTL or structural level, implements the operations described in the data sheet or whatever other specification exists. Verification at the RTL level can be accomplished by means of simulation, but there is a growing tendency to supplement simulation with formal methods such as model checking. At the structural level the use of equivalence checking is becoming standard procedure. In this operation the RTL model is compared to a structural model, which may have been synthesized by software or created manually. Equivalence checking can determine if the two levels of abstraction are equivalent. If they differ, equivalence checking can identify where they differ and can also identify what logic values cause a difference in response.

The emphasis in this chapter is on design verification. When performing verification, the target device can be viewed as a white box or a black box. During *white-box testing*, detailed knowledge is available describing the internal workings of the device to be tested. This knowledge can be used to direct the verification effort. For

example, an engineer verifying a digital circuit may have schematics, block diagrams, RTL code that may or may not be suitably annotated, and textual descriptions including timing diagrams and state transition graphs. All or a subset of these can be used to advantage when developing test programs. Some examples of this were seen in Chapter 9. The logic designer responsible for the correctness of the design, armed with knowledge of the internal workings of the design, writes stimuli based on this knowledge; hence he or she is performing white-box testing.

During *black-box testing* it is assumed that there is no visibility into the internal workings of the device being tested. A functional description exists which outlines, in more or less detail, how the device must respond to various externally applied stimuli. This description, or specification, may or may not describe behavior of the device in the presence of all possible combinations of inputs. For example, a microprocessor may have op-code combinations that are left unused and unspecified. From one release to the next, these unused op-codes may respond very differently if invoked. PCB designers, concerned with obtaining ICs that work correctly with other ICs plugged into the same PCB or backplane, are most likely to perform black-box testing, unless they are able to persuade their vendor to provide them with more detailed information.

Some of the tools used for design verification of ICs have their roots in software testing. Tools for software testing are sometimes characterized as *static analysis* and *dynamic analysis* tools. Static analysis tools evaluate software before it has run. An example of such a tool is *Lint*. It is not uncommon, when porting a software system to another host environment and recompiling all of the source code for the program, to experience a situation where source code that compiled without complaint on the original host now either refuses to compile or produces a long list of ominous sounding warnings during compilation. The fact is, no two compilers will check for exactly the same syntax and/or semantic violations. One compiler may attempt to interpret the programmer's intention, while a second compiler may flag the error and refuse to generate an object module, and a third compiler may simply ignore the error.

Lint is a tool that examines C code and identifies such things as unused variables, variables that are used before being initialized, and argument mismatches. Commercial versions of Lint exist both for programming languages and for hardware design languages. A lint program attempts to discover all fatal and nonfatal errors in a program before it is executed. It then issues a list of warnings about code that could cause problems. Sometimes the programmer or logic designer is aware of the coding practice and does not consider it to be a problem. In such cases, a lint program will usually permit the user to mask out those messages so that more meaningful messages don't become lost in a sea of detail.

In contrast to static analysis tools, dynamic analysis tools operate while the code is running. In software this code detects such things as memory leaks, bounds violations, null pointers, and pointers out of range. They can also identify source code that has been exercised and, more importantly, code that has not been exercised. Additionally, they can point out lines of code that have been exercised over only a partial range of their variables.

12.3 SIMULATION

Over the years, simulation performance has benefited from steady advances in both software and hardware enhancements, as well as modeling techniques. Section 2.12 provides a taxonomy of methods used to improve simulation performance. Nonetheless, it must be pointed out that the style of the code written by the logic designer, as well as the level of abstraction, can greatly influence simulation performance.

12.3.1 Performance Enhancements

Several approaches to speeding up simulation were discussed in Chapter 2. Many of these approaches impose restrictions on design style. For example, asynchronous circuit design requires that the simulator maintain a detailed record of the precise times at which events occur. This is accomplished by means of delay values, which facilitate prediction of problems resulting from races and hazards, as well as setup and hold violations, but slow down simulation.

But why the emphasis on speed? The system analyst wants to study as many alternatives as possible at the conceptual level before committing to a detailed design. For example, the system analyst may want to model and study new or revised op-codes for a microprocessor architecture. Or the analyst may want to know how many transactions a bank teller machine can perform in a given period of time. Throughput, memory and bandwidth requirements for system level designs can all be more thoroughly evaluated at higher levels of abstraction. Completely new applications can be modeled in order to perform feasibility studies whose purpose is to decide how to divide functionality between software and hardware. Developing a high-level model that runs quickly, and coding the model very early in the conceptual design phase, may offer the additional benefit that it can permit diagnostic engineers to begin writing and debugging their programs earlier in the project.

The synchronous circuit, when rank-ordered and using zero delay, can be simulated much more efficiently than the asynchronous circuit, because it is only necessary to evaluate each element once during each clock period. Timing analysis, performed at the structural or gate level, is then used to ensure that path delays do not exceed the clock period and do not violate setup and hold times. Synchronous design also makes it possible to employ compiled code, rather than interpreted code which uses complex tables to link signals and variables. A Verilog or VHDL model can be compiled into C or C++ code which is then compiled to the native language of the host computer. This can provide further reduction in simulation times, as well as significant savings in memory usage, since variables can be linked directly, rather than through tables and pointers.

The amount of performance gain realized by compiled code depends on how it is implemented. The simplest approach, from an implementation standpoint, is to have all of the compiled code execute on every clock cycle. Alternatively, a pseudo-event-driven implementation can separate the model into major functions and execute the

compiled code only for those functions in which one or more inputs has changed. This requires overhead to determine which blocks should be executed, but that cost can be offset by the savings from not executing blocks of code unnecessarily.

The type of circuit being simulated is another factor that determines how much gain is realized by performing rank-ordered, zero delay simulation. In a pure combinational, gate-level circuit, such as a multiplier array, if timing-based, event-driven simulation is performed, logic gates may be evaluated multiple times in each clock cycle because logic events occur at virtually every time slot during that period. These events propagate forward, through the cone they are in, and converge at different times on the output of that cone. As a result, logic gates at or near the output of the cone may be evaluated tens or hundreds of times. Thus, in a large combinational array, rank-ordered, zero delay simulation may realize 10 to 100 times improvement in simulation speed.

Traditionally, point accelerators have been used to speed up various facets of the design task, such as simulation. The use of scan in an emulation model makes it possible to stop on any clock and dump out the contents of registers in order to pinpoint the source of an incorrect response. However, while they can significantly speed up simulation, point accelerators have their drawbacks. They tend to be quite costly and, unlike a general-purpose workstation, when not being used for simulation they stand idle. There is also the risk that if an accelerator goes down for any length of time, it can leave several logic designers idle while a marketing window of opportunity slowly slips away. Also, the point accelerator is a low-volume product, hence costly to update, while the general-purpose workstation is always on an upward spiral, performancewise. So the workstation, over time, closes the performance gap with the accelerator.

By way of contrast, a cycle simulator (cf. Section 2.12), incorporating some or all of the features described here, can provide major performance improvements over an event-driven simulator. As a software solution, it can run on any number of readily available workstations, thus accommodating several engineers. If a single machine fails, the project can continue uninterrupted. If a simulation task can be partitioned across multiple processors, further performance gains can be obtained. The chief requirement is that the circuit be partitioned so that results only need be communicated at the end of each cycle, a task far easier to perform in the synchronous environment required for cycle simulation. Flexibility is another advantage of cycle simulation; algorithm enhancements to a software product are much easier to implement than upgrades to hardware.

It was mentioned earlier that a user can often influence the speed or efficiency of simulation. One of the tools supported by some commercial simulators is the *profiler*. It monitors the amount of CPU time spent in each part of the circuit model being simulated. At the end of simulation a profiler can identify the amount of CPU time spent on any line or group of lines of code. For compute-intensive operations such as simulation, it is not unusual for 80–95% of the CPU time to be spent simulating a very small part of the circuit model. If it is known, for instance, that 5% of the code consumes 80% of the CPU time, then that part of the code can be reviewed with the intention of writing it more efficiently, perhaps at a higher level of

abstraction. Streamlining the code can sometimes produce a significant improvement in simulation performance.

12.3.2 HDL Extensions and C++

There is a growing acceptance of high-level languages (HLLs), particularly C and C++, for conceptual or system level modeling. One reason for this is the fact that a model expressed in an HLL usually executes more rapidly than the same model expressed in an RTL language. This is based, at least in part, on the fact that when a Verilog or VHDL model is executing as compiled code, it is first translated into C or C++. This intermediate translation may introduce inefficiencies that the system engineer hopes to avoid by directly encoding his or her system level model in C or C++. Another attraction of HLLs is their support for complex mathematical functions and similar such utilities. These enable the system analyst to quickly describe and simulate complex features or operations of their system level model without becoming sidetracked or distracted from their main focus by having to write these utility routines.

To assist in the use of C++ for logic design, vendors provide class libraries.[3] These extend the capabilities of C++ by including libraries of functions, data types, and other constructs, as well as a simulation kernel. To the user, these additions make the C++ model look more like an HDL model while it remains legal C++ code. For example, the library will provide a function that implements a wait for an active clock edge. Other problems solved by the library include interconnection methodology, time sequencing, concurrency, data types, performance tracking, and debugging. Because digital hardware functions operate concurrently, devices such as the timing wheel (cf. Section 2.9.1) have been invented to solve the concurrency issue at the gate-level. The C++ library must provide a corresponding capability. Data types that must be addressed in C++ include tri-state logic and odd data bus widths that are not a multiple of 2. After the circuit model has been expressed in terms of the library functions and data types, the entire circuit model may then be linked with a simulation kernel.

An alternative to C++ for speeding up the simulation process, and reducing the effort needed to create testbenches, is to extend Verilog and VHDL. The IEEE periodically releases new specifications that extend the capabilities of these languages. The release of Verilog-2001, for example, incorporates some of the more attractive features of VHDL, such as the “generate” feature. Vendors are also extending Verilog and VHDL with proprietary constructs that provide more support for describing operations at higher levels of abstraction, as well as support for testbench verification capabilities—for example, constructs that permit complex monitoring actions to be compressed into just a few lines of code. Oftentimes an activity such as monitoring events during simulation—an activity that might take many lines of code in a Verilog testbench, and something that occurs frequently during debug—may be implemented very efficiently in a language extension. The extensions have the advantage that they are supersets of Verilog or VHDL; hence the learning curve is quite small for the logic designer already familiar with one of these languages.

A danger of deviating from existing standards, such as Verilog and VHDL, is that a solution that provides major benefits while simulating a design may not be compatible with existing tools, such as an industry standard synthesis tool or a design verification tool. As a result, it becomes necessary for a design team to first make a value judgment as to whether there is sufficient payback to resort to the use of C++ or one of the extension languages. The extension language may be an easier choice. The circuit under design is restricted to Verilog or VHDL while the testbench is able to use all the features of Verilog or VHDL plus the more powerful extensions provided by the vendor.

If C++ is chosen for systems level analysis, then once the system analyst is satisfied that the algorithms are performing correctly, it becomes necessary to convert the algorithms to Verilog or VHDL for implementation. Just as there are translators that convert Verilog and VHDL to C or C++ to speed up simulation, there are translators that convert C or C++ to Verilog or VHDL in order to take advantage of industry standard synthesis tools. The problem with automating the conversion of C++ to an RTL is that C++ is quite powerful, with many features that bear no resemblance to hardware, so it is necessary to place restrictions on the language features that are used, just as synthesis tools currently restrict Verilog and VHDL to a synthesizable subset. Without the restrictions, the translator may fail completely. Restrictions on the language, in turn, place restrictions on the user, who may find that a well-designed block of code employs constructs that are not supported by the particular translator being used by the design team. This necessitates recoding the function, often in a less expressive form.

12.3.3 Co-design and Co-verification

Many digital systems have grown so large and complex that it is, for all practical purposes, impossible to design and verify them in the traditional manner—that is, by coding them in an HDL and applying stimuli by means of a testbench. Confidence in the correctness of the design is only gained when it is seen to be operating in an environment that closely resembles its final destination. This is often accomplished through the use of co-design and co-verification.*

Co-design simultaneously designs the hardware and software components of a system, whereas co-verification simultaneously executes and verifies the hardware and software components. Traditionally, hardware and software were kept at arms length while designing a system. Studies would first be performed, architectural changes would be investigated, and the hardware design would be "frozen," meaning that no more changes would be accepted unless it could be demonstrated that they were absolutely essential to the proper functioning of the product. The amount of systems analysis would depend on the category of the development effort: Is it a completely new product, or an enhancement (cf. Section 1.4)? If it is an enhancement to an existing product, such as a computer to which a few new op-codes are to be added, then compatibility with existing products is essential, and that becomes a

*Co-design and co-verification often appear in the literature without the hyphen—that is, as codesign and coverification.

constraint on the process. A completely new product permits much greater freedom of expression while investigating and experimenting with various configurations.

The co-design process may be focused on finding the best performance, given a cost parameter. Alternatively, the performance may be dictated by the marketplace, and the goal is to find the most economical implementation, subject to the performance requirements. Given the constraints, the design effort then shifts toward identifying an acceptable hardware/software partition. Another design parameter that must be determined is control concurrency. A system's control concurrency is defined by the functional behavior and interaction of its processes.[4] Control concurrency is determined by merging or splitting process behaviors, or by moving functions from one process to another. In all of these activities, there is a determined effort to keep open channels of communication between the software and hardware developers so that the implications of tradeoffs are completely understood.

The task of communicating between diverse subsystems, some implemented in software and some in hardware, or some in an HDL and some in a programming language, presents a challenge that often requires an ad-hoc solution. The flow in Figure 12.1 represents a generic co-design methodology.[5] In this diagram, the hardware may be modeled in Verilog, VHDL, or C++ or it could be modeled using field programmable gate arrays (FPGAs). Specification of the hardware depends on its purpose. Decisions must be made regarding datapath sizes, number and size of registers, technology, and so on.

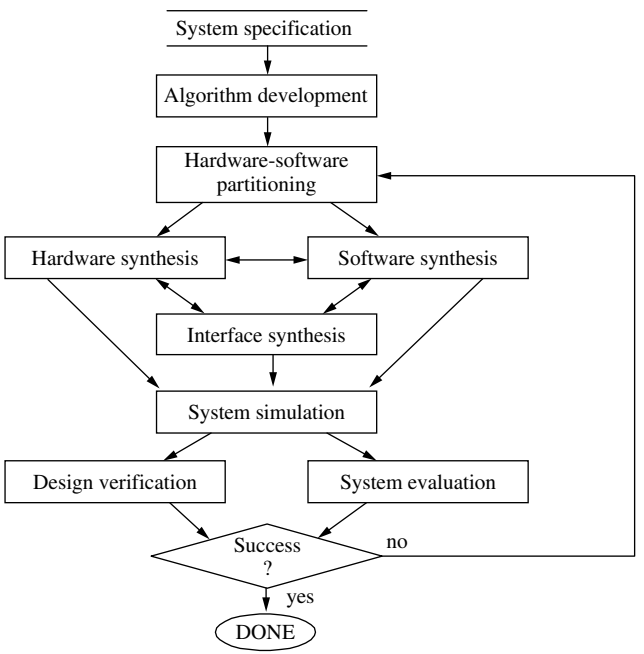


Figure 12.1 Generic co-design methodology.

The interface between hardware and software must handle communications between them. If the model is described in Verilog, running under Unix, then the Verilog programming language interface (PLI) can communicate with software processes using the Unix socket facility. After the design has been verified, system evaluation determines whether the system, as partitioned and implemented, satisfies performance requirements at or under cost objectives. If some aspect of the design falls short, then another partitioning is performed. This process can be repeated until objectives are met, or some optimum flow is achieved. Note that if the entire system is developed using C++, many communications problems are solved, since everything can be compiled and linked as one large executable.

12.4 MEASURING SIMULATION THOROUGHNESS

As indicated previously, many techniques exist for speeding up simulation, thus permitting more stimuli to be applied to a design in a given period of time. However, in design verification, as in manufacturing test, it is important not to just run a lot of stimuli, but also to measure the thoroughness of those stimuli. Writing stimuli blindly, without evaluating their effectiveness, may result in high quantities of low-quality test stimuli that repeatedly exercise the same functionality. This slows down the simulations without detecting any new bugs in the design. Coverage analysis can identify where attention needs to be directed in order to improve thoroughness of the verification effort. Then, the percentage coverage of the RTL, rather than the quantity of testbench code, becomes the criteria for deciding when to bring design verification to a halt.

12.4.1 Coverage Evaluation

Chapter 7 explored a number of topics, including toggle coverage (Section 7.8.4), gate-level fault simulation (Section 7.5.2), behavioral fault simulation (Section 7.8.3), and code coverage (Section 7.8.5). Measuring toggle coverage during simulation was a common practice many years ago. It was appealing because it did not significantly impact simulation time, nor did it require much memory. However, its appeal for design verification is rather limited now because it requires a gate-level model. If a designer simulates at the gate level and finds a bug, it usually becomes necessary to resynthesize the design, and designers find it inconvenient to interrupt verification and resynthesize each time a bug is uncovered, particularly in the early stages of design verification when many bugs are often found in rapid succession. As pointed out in Section 7.8.4, toggle count remains useful for identifying and correcting hot spots in a design—that is, areas of a die that experience excessive amounts of logic activity, causing heat buildup. It was also argued in Chapter 7 that fault simulation can provide a measure of the thoroughness of design verification vectors. But, like toggle count, it relies on a gate-level model.

Code coverage has the advantage that it can be used while simulating at the RTL level. If a bug is found, the RTL is corrected and simulation continues. The RTL is

not synthesized until there is confidence in the correctness of the RTL. As pointed out in Section 7.8.5, code coverage can be used to measure block coverage, expression coverage, path coverage, and coverages specific to state machines, such as branch coverage. When running code coverage, the user can identify modules of interest and omit those that are not of interest. For example, the logic designer may include in the design a module pulled down from a library or obtained from a vendor. The module may already have been thoroughly checked out and is currently being used in other designs, so there is confidence in its design. Hence it can be omitted from the coverage analysis.

Code coverage measures controllability; that is, it identifies all the states visited during verification. For example, we are given the equation

$$WE = CS \ \& \ ArraySelect \ \& \ SectorSelect \ \& \ WriteRequest;$$

What combinations of the input variables are applied to that expression? Does the variable *SectorSelect* ever control the response of *WE*? In order for *SectorSelect* to control *WE*, it must assume the values 0 and 1 while the other three inputs must be 1. For this expression, a code coverage tool can give a coverage percentage, similar to a fault coverage percentage, indicating how many of the variables have controlled the expression at one time or another during simulation. Block coverage, which indicates only whether or not a line of code was ever exercised, is a poor measure of coverage. When verifying logic, it is not uncommon to get the right response for the wrong reason, what is sometimes referred to as *coincidental correctness*. For example, two condition code bits in a processor may determine a conditional jump, but the one that triggered the jump may not be the one currently being investigatated.

Consider the state machine: It is desirable to visit all states, and it is desirable to traverse all arcs. But, in a typical state machine several variables can control the state transitions. Given a compound expression that controls the transition from S_i to S_j, a thorough verification requires that each of the variables, at some point during verification, causes or determines the transition to S_j. In general, equations can be evaluated to determine which variables controlled the equation and, more importantly, which variable never controlled the equation throughout the course of simulation. An important goal of code coverage is to verify that the input vectors established logic values on internal signals in such a way that the outcome of a logic transaction depends only on one particular signal, namely, the signal under consideration.

Behavioral fault simulation, in contrast to code coverage, measures both controllability and observability. A fault must be sensitized, and its effects must be propagated to an observable output before it can be counted as detected. One drawback to behavioral fault simulation is the fact that the industry has never settled on an acceptable family of faults, in contrast to gate-level fault simulation where stuck-at-1 and stuck-at-0 faults have been accepted for more than a quarter-century.

Given a fault coverage number estimated using a gate-level model, test engineers can usually make a reasonably accurate prediction of how many tester escapes to

expect from their product lines. So, although the stuck-fault metric is not perfectly accurate, it is a useful tool for estimating outgoing quality level. Furthermore, many studies over the years have helped to refine our understanding of the various gate-level fault models. For example, it is well known that fault models based on stuck-at faults in gate-level circuits can produce widely divergent results, depending on which faults are selected and how the fault list is collapsed. Many years ago it was shown that vectors providing a coverage of 95% for pin faults on SSI and MSI circuits provided in the neighborhood of 70–75% fault coverage when internal faults were considered.[6,7]

Another drawback to the use of behavioral fault simulation for design verification is the fact that it only counts as detected those faults that propagate to the output pins. For design verification, it is frequently unnecessary to propagate behavioral faults to an output pin, it is sufficient to sensitize (i.e., control) the faults. But, as we have just seen, code coverage measures controllability, and its metrics are well understood and accepted. So, if the goal is simply to sensitize logic, then code coverage is adequate.

Another means for determining the thoroughness of coverage is through the use of event monitors and assertion checkers.[8] The *event monitor* is a block of code that monitors events in a model in order to determine whether some specific behavior occurred. For example, did the applied stimuli try to write to a fifo when it was full? This is a situation that will occur in practice; and in order to determine if the circuit responds correctly, it is necessary to first verify that this condition occurred and then verify that the circuit responded as desired. One way to check for this condition is to write a block of code that checks for "fifo full" and "write enabled." The code can be embedded conditionally into a Verilog RTL model using <"ifdef", "endif"> pairs, or it can be coded as a standalone module. If the conditions "*fifo_full*" and "*write_request*" are both found to be true, a message can be written to a log file and the engineer can then check the circuit response to verify that it is correct.

The *assertion checker* is implemented like an event monitor, but it is used to detect undesirable or illegal behavior. Consider the case of a circuit that is required to respond within 50 clock periods to a bus request. This is classified as a temporal assertion, because the event is required to occur within a specified time interval, in contrast to the previous example of the fifo, which is classified as a static event—that is, one in which the events occur simultaneously. It would be tedious to enumerate all of the possible cases that should be checked during simulation, but many corner cases can be defined and monitored using monitors and checkers.

Monitors and checkers can supplement code coverage as a means of measuring the thoroughness of a test suite. If there are specific corners of a design that the designer is interested in, monitors and checkers can explicitly check those cases. A response from the appropriate checker can put the logic designer's mind at ease. It might, however, be argued that if the logic designer used code coverage and obtained 100% expression coverage, and verified that the circuit responded correctly for all stimuli, then the designer has already checked the condition.

Example Consider the fifo example cited earlier. Somewhere in the logic there may be an expression similar to the following:

$$mem\_avail = fifo\_full \ \& \ write\_request;$$

In this expression *fifo_full* is high if the fifo is full, and it is low otherwise. *Write_request* goes high if an attempt is made to write to the fifo. If memory is available, *fifo_full* is low and *mem_avail* is low. However, if an attempt is made to write to the fifo when it is full, *mem_avail* goes high. If code coverage confirms 100% coverage for this line of code, then all possibilities have been checked. The following is a table of results that might be printed by a code coverage tool.

Count	*fifo_full*	*write_request*	*mem_avail*
3243	0	1	0
3	1	0	0
0	1	1	1
66%		Expression coverage	

■■

These code coverage results indicate that no write requests were attempted when the fifo was full (count = 0). An advantage of monitors and checkers over code coverage is that they check for specific events that the logic designer is concerned about, so the designer does not have to scroll through a large file filled with detail. In addition, code coverage only checks for controllability. The event monitor can be coded and positioned in the model in such a way as to confirm complete transactions, including events occurring at the memory and at the destinations. However, regardless of which method is used, in the final analysis the logic designer must understand the design and verify that the design implements the specification, rather than his subjective interpretation of the specification.

12.4.2 Design Error Modeling

While the use of behavioral fault simulation for design verification may be of questionable value, it can be useful for evaluating a manufacturing test suite prior to synthesis. The granularity is more coarse than that of the gate-level model, but it may nevertheless point to areas of a design where coverage is particularly weak and where design changes might be helpful. For example, controllability may be quite poor because long input sequences are needed to reach a particular state, suggesting that perhaps a parallel load of some counter may be desirable. Perhaps an unused state in a state machine can be used to load a particular register in test mode in order to improve controllability. Or this unused state may be used to gate test data out onto a bus, thus improving observability. By including such changes at the RTL level, in response to low behavioral fault coverage, the changes can be evaluated and verified before the circuit is synthesized. Behavioral fault simulation can also be useful in evaluating diagnostic programs that are intended to be run in the field.

In earlier chapters it was noted that if a fault was modeled and detected by a fault simulator, we can expect it to be detected when the chip is tested. However, fault simulation cannot say anything about faults that are not modeled. In like manner, design verification can confirm the correctness of operations that are exercised by the applied vectors, but it cannot prove the absence of design errors in functions that were not targeted by the vectors.

This is important to note because, even for very small circuits, the number of potential errors becomes impractical to consider. In Section 7.7.1 an example was given wherein, for a simple two-input circuit, 16 possible functions were defined. For a complex sequential circuit with n inputs and m internal states, the number of potential states becomes astronomical very quickly. The task of counting the exact number of states is further exacerbated by the fact that many of the states are unreachable in incompletely specified state machines (ISSMs). Furthermore, it is not immediately obvious how many state transitions are required to reach a given state from some other, arbitrary state. At best, all we can hope to do is compute an upper bound on the number of clock cycles required to completely exercise a given sequential circuit. The reader may recall, from Section 3.4, that these considerations led early researchers dealing with manufacturing test to introduce the concept of a stuck-at-fault.

Faster simulation methodologies, such as cycle simulation and point accelerators, have been introduced in order to improve thoroughness of design verification. In this approach, logic designers keep doing what they have done in the past, but they do it faster and they do more of it, in the hopes that by using more stimuli they will be more thorough. The problem with this method is that, like manufacturing test programs, if there is no way to evaluate the thoroughness or completeness of the programs, it is possible to quickly reach the point of diminishing returns: Many thousands of additional vectors are added without improving the overall thoroughness of the verification effort. Author Boris Beizer calls it the "pesticide paradox," wherein insects build up a tolerance for pesticides, and the continued application of these same pesticides does not remove any more insects from the fields.[9]

The stuck-at model has been an accepted metric for over three decades. While it is recognized that it is not perfect, it is understood that if stuck-at coverage for a manufacturing test is 70%, there will be many tester escapes. If stuck-at coverage is greater than 98%, the number of tester escapes is likely to be very low. Software analysts have used error seeding to compute a similar number. This involves the intentional insertion or errors in a design. The design error coverage C_{DE}, analogous to fault coverage, is

$$C_{DE} = \frac{\text{number of errors detected}}{\text{number of errors injected}} * 100\%$$

The C_{DE} might be determined by having one group inject design errors and another independent group write design verification suites. Just as the fault coverage based

on stuck-at faults is not perfect, the design error coverage, based on injected faults, may be either too optimistic or too pessimistic. However, if $C_{DE} = 70\%$, it is a good idea to keep on writing design verification vectors. If $C_{DE} = 100\%$ and if no bugs have been encountered in some arbitrary interval (e.g., 1 week), then considerable thought must be given to deciding whether the device is ready to be shipped, recognizing that even if $C_{DE} = 100\%$, it only guarantees that all of the artificially created and injected design errors were detected, there may still be real errors in the design.

If error seeding is to be used, it must be decided what kind of errors to inject into the circuit model. In view of the fact that contemporary circuits are designed and debugged at the register transfer level, errors should be created and injected at that level. Like fault simulation, granularity is an issue to consider. Stuck-at faults can cause detection of gross physical defects in addition to stuck-at faults. In like manner, gross design errors (e.g., a completely erroneous algorithm implementing arithmetic/logic operations) are likely to be detected by almost any verification suite, so it makes sense to inject subtle errors that are more difficult to discover. This includes such things as wrong operators in RTL expressions, incorrect variables, or incorrect subscripts. For example, consider the following Verilog expression:

always @(*sign* or *a* or *b* or *c* or *d* or *e*)

g = (!*sign*) ? *a* | !(*b* | *c*) & *d* | !*e* : 0;

If *sign* is equal to 0, the complex expression is evaluated and its value is assigned to *g*; else 0 is assigned to *g*. Some very simple errors that can be applied to this Verilog code include leaving out a negation (!) symbol, or placing a left or right parenthesis in the wrong place, or substituting an OR (|) for an AND (&) or vice versa. One of the terms might be modified by adding a variable to the product. Sometimes the failure to include a variable in the sensitivity list, particularly if it is a long list, can cause a logic designer to puzzle for quite some time over the cause of an erroneous response in an equation that appears well-formed.

The misuse of blocking and non-blocking assignments in Verilog procedural statements can cause confusion. Blocking assignments, indicated by the symbol (=), can suspend, or block, a process until a register is updated. A non-blocking assignment, indicated by the symbol (<=), permits a register to be evaluated, but updated at a later time, while permitting processing to continue, hence the term non-blocking.

For more complex expressions, such as loop control, error injection can consist of changing limits, or polarity of a control signal. In case statements intended to represent state machines, incorrect state machine behavior can be induced by switching cases. More difficult to detect is the situation where, in one of the cases, a complex expression is altered. In effect, a good design verification suite should exhaustively consider all possible values of the variables in a complex expression. This is equivalent to having 100% expression coverage for the expression from a code coverage tool. Altering the order of the variables in a port list may also provide a good challenge for a design verification suite.

If seeding of design errors can be accomplished by a program, similar to fault list generation for gate-level fault simulation, some of the subjectivity that causes potential errors to be overlooked can be eliminated. The human may make a judgment as to whether or not it is necessary to seed a particular part of a design, or to use a particular error construct. The program, on the other hand, seeds according to some predetermined formula. The subjectivity of the design verification process is also a good reason why a design verification suite is best developed by individuals other than those who designed the circuit. It also explains why software code inspections are performed by persons other than those who wrote the software. It is not uncommon for someone who wrote a block of code, whether it be HLL or HDL, to examine that code several times and not see an obvious error. A similar situation holds for a specification. The designer may misunderstand some fine point in the specification and, if he creates stimuli based on this misconception, his simulation results only confirm that his design functions according to his understanding, which was initially wrong.

A typical practice when testing S/W is to inject bugs one at a time. After a run has completed, S/W responses with and without the injected bug are compared. If the injected bug causes incorrect response, it has been detected. It is not necessary to debug the circuit since the bug was injected; hence its location is known. Of course, if the bug escapes detection, then it becomes necessary to determine why it was not detected. In a regression test, a bug that was previously detected may now escape detection as a result of a patch inserted to fix another bug. Design error injection in HDL designs is quite similar to S/W testing. One noticeable difference is the fact that response of an HDL can be examined at I/O pins. But, recalling our previous discussion, logic designers may choose not to drive an internal state to an I/O pin. Hence it may be necessary to capture internal state at registers and state machines and then output that information to a file where it can be checked for correctness.

12.5 RANDOM STIMULUS GENERATION

In previous sections we explored methods for simulating faster, so more stimuli could be evaluated in a given amount of time, and we explored methods for measuring thoroughness of design verification stimuli. A report generated during coverage analysis identified modules or functions where coverage was insufficient. We now turn to stimulus generation. In this section we focus on random stimulus generation. In subsequent sections, we will explore behavioral automatic test pattern generation.

One of the purposes of test stimuli created and applied to a design is to give us confidence in the correctness of the design. The more functionality we verify, the greater our confidence. Unfortunately, confidence is a subjective thing. We may feel 100% confident in a design that has only been 80% verified! For example, in a survey, circa 1990, of IC foundries that fault-simulated stimuli provided by their customers, it was found that a typical test suite provided by customers yielded

approximately 73% fault coverage for stuck-at faults in the IC. These test suites were developed during design verification and served as the acceptance test for ICs provided by the foundry. Part of the reason for low coverage stems from decisions by logic designers regarding the importance of verifying various parts of the design. It is not uncommon for a logic designer to make subjective decisions as to which parts of a design are "complicated" and need to be thoroughly checked out, based on his or her understanding of the design, versus those parts of the design that are "straightforward" and need less attention.

Random test pattern generation (RTPG) is frequently used to exercise designs. Unlike targeted vectors, random vectors distribute stimuli uniformly across the design, unless some biasing is built into the vectors (cf. Section 9.4.3, weighted random patterns).

Given a sufficiently large set of random values and an unbiased set of I/O pins, each input combination is equally probable. Given a combinational array implementing arithmetic operations, it is often quite easy to create a configuration like that of Figure 12.2 for an ALU or similar such circuit.

The random pattern generator (RPG) generates a pair of n-wide integers. These are simulated using the circuit model, but the result is also computed independently of the simulation. The results are then sent to a comparator that translates the integer result into binary and compares the two results in order to determine whether the design responded correctly. The whole process can be automated, and the number of stimuli applied to the design is limited only by the speed of the simulation process. A typical stopping rule for such a process is to cease testing when no more errors are detected after some predetermined number of stimuli have responded correctly.

For sequential circuits, RTPG is a more difficult task because circuit response depends on current state of the circuit. For example, if a chip-select is disabled, no amount of stimuli applied to the other input pins will serve a useful purpose until the chip-select is enabled. Even if the chip-select is enabled, stimuli on other input pins may be ineffective if an internal control register has not been initialized. But even a fully initialized circuit may recognize only a small number of input combinations from its current state. A microprocessor, for example, may be in a state for which only a single-input combination is useful. Such an example might be a hold or a halt instruction, for which a controlling state machine only responds to a valid interrupt request.

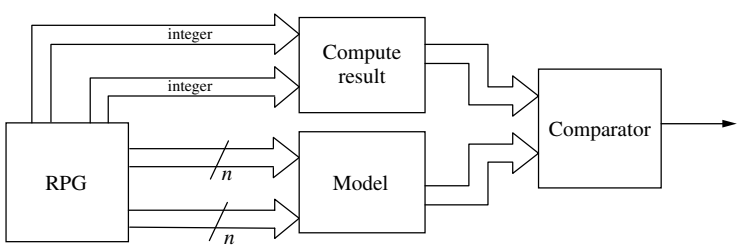


Figure 12.2 Applying random stimuli.

Another complication is the fact that contemporary microprocessors employ multiple pipelines to decode instructions and allocate resources needed to successfully execute those instructions. Out-of-order execution of instructions, and contention for resources by instructions being decoded and executed in parallel pipelines, means that priorities have to be resolved. If two instructions being decoded in different pipelines both require the same general-purpose register, which instruction gets to use it first? Because of out-of-order execution, an op-code may attempt to perform an operation on a register whose value has not yet been set.

Clearly, in these complex processors, it is necessary to exercise every instruction with all combinations of meaningful data. Load instructions should point at memory addresses containing valid data. Branch instructions must have valid instructions at the branch address, and the test must be configured so as to avoid infinite loops. Conditional branches must be exercised with all condition codes and combinations of condition codes. Furthermore, it must be verified that branches can be made to occur, or inhibited, depending on the settings of the condition codes.

Testing the interrupt structure means not just testing for correct operation of individual interrupts, but also testing to ensure that correct priorities are observed. If an interrupt is being processed and another interrupt occurs, is the new interrupt of higher or lower priority than the interrupt currently being processed? If it is of higher priority, then current interrupt processing must be interrupted, and the new interrupt must be processed; then the processor must resume processing the interrupt that was originally being processed. In addition to the interrupt inputs, other input pins must also be exercised at the appropriate times to determine their effect on the behavior of the design. This includes chip select pins, memory and I/O read and write pins, and any other pins that are able to affect the flow of control in the design.

In a program for generating test suites for microprocessors described at the 1982 Design Automation Conference,[10] the various properties of the microprocessor were systematically captured in a file. This included information about instruction formats, register file sizes, ALU operations, I/O pins, and their effects on the flow of instructions and data. Details of addressing methods and formats included descriptions of program counters, index registers, stack pointers, and relative and absolute addressing methods. In addition, information describing controllability and observability methods of the registers was provided to the system. With this information, the automatic generation system synthesized sequences of instructions, including the necessary initialization sequences. Where the system might generate an excessive number of instructions—as, for instance, when generating sequences that test every register combination for a move register instruction—the user had the option of selecting a subset adequate to satisfy the objectives of the test.

In another method, whose purpose was to verify the design of an original version of an IBM System/6000 RISC processor, RTPG was used to make the test program generation process more productive, comprehensive, and efficient.[11] The system developed was a dynamic, biased pseudo-random test program generator. Unlike a so-called static approach where a test program was developed and then simulated in its entirety, the RTPG system developed by this project was dynamic: Test

generation was interleaved with the execution of instructions. This made it possible for the logic designer to create test programs during the early stages of design, while implementing the op-codes.

The test program generated by RTPG is made up of three parts:

Initial state
Instructions
Expected results

The initial state specifies the contents of resources needed to execute a particular instruction, including registers, flags, and memory contents. Instructions describe the contents of caches or memory locations. Expected results list the final state of all resources that were affected by the execution of the instruction. These test programs are self-contained and include all information required for their independent execution, so they can migrate between test libraries and they can be executed in any order.

Example

```
H 10000:
* A simple test program
R IP        00010000
R R1        03642998
R R8        0000000F
R R10       1E12115F
R R22       0129DFFF
R R30       800000BA
R MSR       00008000
R CR        8CC048C8
R XER       2000CD45
D 0129DFFC 4E74570E
D 03640B90 7D280411
* ------ Assembly Program -------------
I 00010000 7C48F415   a0.  R2 .R8.R30
I 00010004 7CD0B02E   lx   R7 .R0.R22        E/A 0129DFFF
I 00010008 49BBB904   b    *+29079812        T/A 01BCB90C
I 01BCB90C B141E1F8   sth  R10.X'E1F8'(R1)    E/A 03640B90
*------- Expected Results -------------
R IP        01BCB910
R R2        800000C9
R R7        4E74570E
R MSR       00008000
R CR        8CC048C8
```

```
R XER       0000CD45
D 0129DFFC  4E74570E
D 03640B90  115F0411
END                                                    ■■
```

In this example the header (H) is used to identify the test number. The next line, starting with an asterisk, denotes a comment. The lines beginning with R denote registers. The instruction pointer (IP) identifies the start of the test program—in this case, hex location 10,000. The data entries (D) define memory locations and the data stored at those locations. The instruction (I) entries identify memory addresses and the instructions to be saved at those locations. Note that the first three instructions are contiguous, and then the fourth entry is some distance away from the previous three. The instructions contain assembly code for documentation purposes. In this sequence, the instructions sequence contains add, load, branch, and store instructions. The third instruction causes a branch to location 01BCB90C, where the store instruction is located.

The short program in this example can be executed as soon as all of the instructions used in the example have been implemented. The RTPG initializes the registers used by the instructions being tested, so it is not necessary to employ load and store instructions. The RTL language used for this project was APL (a programming language), and the tools are in-house proprietary tools. The test is constructed dynamically, meaning that for each instruction there is a generation stage and an execution stage. During the generation stage an instruction is chosen and required resources are initialized. The execution stage is then invoked to execute the instruction and update affected resources.

Biasing is used in this system to increase the probability of occurrence of events that might otherwise not occur. Biasing directs the generation process toward selected design areas so that most events are tested when the number of test programs is reasonably large. Biasing functions are used to influence the selection of instructions, instruction fields, registers, addresses, data, and other components that go into construction of a test program. Each instruction or process, such as an interrupt or address translation, is represented by a block diagram composed of decision and execution blocks. In every decision block the data affecting the decision are selected in such a way that subsequent blocks are entered with user-specified or RTPG controlled probability. As an example, the user may request that there be a 10% probability that the arguments selected for a floating point operation produce an overflow.

The biasing functions evolve over a number of projects, so weaknesses observed in the RTPG can be corrected by altering the probabilities; consequently, the functions can be influenced by those probabilities. Code coverage techniques can be used to evaluate the behavior of RTPG; and, by identifying weaknesses, such as lines of code not touched by the RTPG, the results of code coverage can be used to improve the biasing functions. Biasing can also be improved by analyzing the effects of fault injection. Faults or design errors are injected into the model, and it is determined whether or not they are detected by any randomly generated test

program. If, at the conclusion of the design verification effort, there are injected errors that went undetected, then either the biasing functions need to be refined, or, perhaps, the circuit requires a greater number of test programs in order to detect all errors.

In yet another project employing RTPG, the object of the effort was a multiprocessor workstation cache controller.[12] The workstations can contain up to 12 processor boards, with each processor board containing three custom VLSI chips and a 128-kbyte cache memory. Main memory is shared among the workstations. One of the chips is a cache controller whose purpose is to make memory transparent to the processors. It manages the cache and communicates with main memory and peripherals. It consists of a processor cache controller (PCC) and a snooping bus controller (SBC). Each of these two subsystems is complex in and of itself, with many states and transitions. When interactions between PCC and SBC are considered, there are many thousands of possible interactions.

Although the object of this verification effort was to verify the cache controller, it was believed that simulating the cache controller by itself would not be sufficient to verify the system's design. So, the simulation model consisted of all three chips, the cache controller, the CPU, and the floating-point coprocessor. However, for the random tester, a stub module replaced the CPU, simplified inside but accurately modeling the interface. This model was easier to write than a full model, it allowed for more flexible timing, and it ran faster than a full model. Three copies of the three-chip workstation model were instantiated in order to verify the memory design.

The stub CPU generated memory references by randomly selecting from a predetermined script. The scripts, an example of which is illustrated in Figure 12.3, consist of action/check pairs, in which the action produces a state change and the check verifies that the change happened correctly. For example, an action might write a particular value to a memory address. The corresponding check verifies that the update occurred correctly, or signals an error if it did not. Because of the random sequencing, an arbitrary amount of time and a random permutation of other actions and checks may elapse between an action and its corresponding check.

CacheOp	Address	Data	Mode
Action			
Write32	0x00000660	0x05050505	User
Check			
Read32	0x00000660	0x05050505	Kernel
Write32	0x00000660	0x05050505	Kernel
End			
Action			
TestSet	0x0000A800	0x0	User
Check			
Read32	0x0000A800	0x1	User
Write32	0x0000A800	0x0	User
End			

Figure 12.3 Action/check pair.

In Figure 12.3 the words Action, Check, and End are keywords that delineate an action/check pair. An entry identifies a cache operation, the cache address, the data to be written to or read from that address, and the mode. Reserved data words can be used to instruct the CPU to expect specific exception conditions, such as a page fault, to occur. In the second action/check pair, the TestSet cache operation expects the current value at address 0x0000A800 to be 0. It then sets the value to 1. A check performed later expects a 1, and then it clears the value so the next execution of the action will find a 0.

The RTPG was determined by its implementers and users to be a major success. Before it was implemented, several months were spent writing design verification tests in assembly language. These tests covered about half of the uniprocessor cases and none of the multiprocessor cases. The initial version of the random tester, written in a week, immediately revealed numerous errors, including significant design problems. The final version of the RTPG required about two months and detected over half the bugs uncovered during functional verification. The strategy devised for the RTPG was to run until it uncovered a problem, or forever if it could not find any. During the early stages the RTPG would run for about 20 minutes on a Sun3/160 workstation. By the end of verification, it had run continuously for two weeks on multiple computers, using different random seeds.

12.6 THE BEHAVIORAL ATPG

The goal of behavioral ATPG (BATG) is to exploit knowledge inherent in RTL and behavioral level circuit descriptions. ATPG programs have traditionally relied on gate-level circuit descriptions; as circuits grew larger, the ATPGs frequently became entangled in a myriad of details. Managing gate-level descriptions for larger circuits requires exorbitant amounts of memory and CPU time. By exploiting behavior rather than structure, and taking advantage of higher levels of abstraction, the amount of detail is reduced, permitting more efficient operation. Perhaps more importantly, it is possible to distinguish between legal and illegal behaviors of state machines, handshaking protocols, and other functions. It is possible to recognize state-space solutions that would be next to impossible to recognize at the gate level. In addition, it becomes possible to recognize when a solution does not exist, and cease exploring that path.

12.6.1 Overview

A simple example of a circuit where behavioral knowledge can be used to advantage is the one-hot encoding of a state machine (see, for example, Figure 9.30). A gate-level ATPG, attempting to justify an assignment to the state machine, may spend countless hours of CPU time trying to justify a logic 1 on two or more flip-flops when the implementation only permits a single flip-flop to be at logic 1 at any given time. By abstracting out details and explicitly identifying legal behavior of the state machine, this difficulty can be avoided.

In other cases the amount of CPU time required to generate a test at the gate level, even when a test exists, is prohibitive. A circuit as basic as an 8-bit binary counter, capable of counting from 0 to 255, can frustrate an ATPG, since it may require as many as 256 time frames to propagate or justify primitive D-cubes of failure (PDCF). In combinational logic a 64- or 80-bit array multiplier represents a significant challenge to a combinational ATPG, even though theory assures us (Section 4.3) that the ATPG, if allowed to run indefinitely, will eventually find a solution. Note that incremental improvements in ATPG performance have been realized by introducing slightly larger primitives, such as 2-to-1 multiplexers and adders, as primitives. This is a rather small concession to the need for a higher level of modeling.

12.6.2 The RTL Circuit Image

Chapter 2 introduced a circuit model in the form of a graph in which nodes corresponded to individual logic elements and arcs corresponded to connections between elements. The nodes were represented by descripter cells containing pointers and other data (see Figure 2.21). The pointers described I/O connections between the output of one element and the inputs of other elements. The ATPG used the pointers to traverse a circuit, tracing through the interconnections in order to propagate logic values forward to primary outputs and justify assignments back toward the inputs.

For logic elements in an RTL circuit the descripter cells bear a similarity, but functions of greater complexity require more entries in the descripter cell. In addition, linking elements via pointers is more complex. In gate-level circuits the inputs of logic gates are identical in function, but in RTL circuits the inputs may be busses and can serve much more complicated functions. The circuit in Figure 12.4 represents a generic view of a function. It is characterized by the fact that its inputs are control and data ports, and its outputs are status and data ports. Furthermore, each of its ports may be n_i bits wide ($n_i \geq 1$) and, when $n_i > 1$, it is important to indicate whether the high-order bit is numbered bit 0 or bit $n_i - 1$. Not shown in this generic model are internal registers. The registers may hold data or control bits.

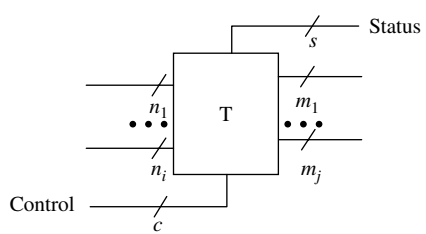


Figure 12.4 Generic representation of a function.

In in the case of a 2-to-1 multiplexer the control could require one or two inputs. One control bit selects one of two data inputs, and the other control bit, if present, enables the output. If the output is disabled, it may be floating (Z state), or forced to a 1 or 0. In the case of an ALU, an operation may require one of several functions to be chosen, thus requiring several control bits. A connectivity graph must embrace all of this information in some orderly way that can be used by many software routines.

When a gate-level ATPG program is implemented, one of the first questions that must be addressed is that of support for primitives. What primitives will the ATPG support? Will the knowledge for these primitives be built into the ATPG, or will that knowledge be represented in tabular form? For example, an AND gate is a primitive for which the ATPG has processing capability. The ATPG may have a routine that simply retrieves the input count for the AND gate and then loops on input values in order to compute the output. When justifying a 0 on the output, it selects one of the inputs and assigns a 0 to the gate driving that input. When propagating through an input, the ATPG knows that it must justify 1s on all the other inputs.

An alternate approach is to employ a truth table, from which PDCFs and other information can be compiled and retrieved as needed (see Section 4.3). An advantage of this is that new primitives can be easily supported simply by adding the appropriate truth table whenever it is advantageous to do so. For example, if a circuit contains many 2-to-1 multiplexers, it may be advantageous to represent the multiplexer as a single primitive, rather than as several logic gates. A standard cell library may have an ATPG model for the multiplexer. When backtracing the 2-to-1 multiplexer using the truth table, the ATPG tries to find an entry in the table that is compatible with the existing state of the circuit. There is no explicit awareness that the multiplexer is making a choice, by way of its control input, from one of two inputs D_0 or D_1.

12.6.3 The Library of Parameterized Modules

For RTL functions, not only are data structures more complex, but processing is also more complex. The types of functions is seemingly endless. How is it possible to create something analogous to a gate-level ATPG? One way to control the scope of the problem is to require that a behavioral ATPG restrict itself to synthesizable circuits. Another way to reduce the scope of the problem, when parsing an RTL circuit, is to identify basic functions and map these into canonical forms. Then the interconnection of these elements is accomplished through pointers, just as is done at the gate level. A logical question to ask is, "How many basic functions are there?" The Electronic Design Interchange Format (EDIF) webpage[13] contains a Library of Parameterized Functions (LPM), which lists 25 basic functions:

CONST	INV	AND	OR	XOR
LATCH	FF	SHIFTREG	RAM_DQ	RAM_IO
ROM	DECODE	MUX	CLSHIFT	COMPARE
ADD_SUB	MULTIPLER	COUNTER	ABS	BUSTRI
FSM	TTABLE	INPAD	OUTPAD	BIPAD

Some of these are obvious, others are not so obvious. The CONST model returns a constant value. CLSHIFT is a combinatorial shifter. RAM_IO has a bidirectional data port, while RAM_DQ has an input data port and an output data port. TTABLE is a truth table and FSM is a finite-state machine.

Each of these entries is characterized by a number of parameters. The following are some of the properties that characterize COUNTER:

Counter width
Direction (up, down, or dynamic)
Enable (clock or count)
Load style (synchronous or asynchronous)
Load data (variable or constant)
Set or clear (synchronous or asynchronous)

If dynamic count is specified, then the direction of count, up or down, is under control of an input pin. There are other properties that need to be considered. For example, the *width* of the counter may be eight bits, but the maximum count of the counter may be less than 2^{width}. If a data structure exists for COUNTER that supports all of the LPM properties, then a counter that appears in an RTL description can be represented by that data structure. If a particular property does not appear in the RTL description, then that field in the data structure is either left blank or set to a default value. A particular counter in a circuit may have a load capability but may not have a set or clear. In such a case the counter can be loaded with an all-0s or all-1s value to implement the set or clear operation.

Some of the entries, including the truth table, the finite-state machine, and the RAM and ROM modules do not have a standard size. A RAM may be a small bank of registers, or it could be a large cache memory. So, in addition to holding parameters that characterize functionality of these devices, the data structure will need to have variably sized data fields that hold the actual data. Memory for a truth table and transition tables for an FSM can be allocated while the circuit model is being constructed, but memory for the RAM and ROM may have to be allocated dynamically.

Recognizing the presence of an LPM function in an RTL circuit description is accomplished by recognizing keywords and commonly occurring expressions. In Verilog the posedge and negedge keywords identify flip-flops. A case statement could represent a multiplexer, or it could represent a state machine (cf. Figure 9.30). The presence of posedge or negedge helps to distinguish between the multiplexer and state machine. A construct such as a counter is detected by observing the counter being incremented or decremented by a constant value. The b16ctr model, a 16-bit counter (see also Section 7.8.2), illustrates the increment operation.

```
module b16ctr(ctrout,din,clk,loadall,incrcntr,
        decrcntr,rst);
parameter width = 32;
output  [width-1:0] ctrout;
```

```
input   [width-1:0] din;
input  clk, rst, loadall, incrcntr, decrcntr;
reg [width-1:0] ctrout;
wire load = loadall & rst;
always @(posedge clk) begin
  if(!load)
      ctrout <= din;
  else if(incrcntr | decrcntr)
    ctrout <= (decrcntr) ? ctrout - 1 : ctrout + 1;
end
endmodule
```

The data width is set to 32, but it can be overridden by the invoking module, so this model could represent a counter of any size. This example always increments or decrements by 1. The increment value could also be a parameter or variable. For example, if this were a program counter, the increment value might be 1, 2, or 4, depending on whether it is incrementing by one byte, a 16-bit word, or a double word. Also it must be noted that a set or reset input may be active low or active high. The clock also may be positive- or negative-edge triggered. These distinctions must be noted and recorded as part of the characterization of the counter.

An *if ... else* construct indicates the presence of a multiplexer. The following Verilog expression describes a 2:1 multiplexer:

```
wire outx = (sel == 1) ? A : B;
```

If the multiplexer has more than two choices, it might be expressed by means of a case statement. A decoder can also use a case statement. A typical decoder expression may appear as follows:

```
case ({I1,I0})
  2'b11: Y[3:0] = 4'b1000;
  2'b10: Y[3:0] = 4'b0100;
  2'b01: Y[3:0] = 4'b0010;
  2'b00: Y[3:0] = 4'b0001;
endcase;
```

When a behavioral ATPG parses an RTL model and associates RTL constructs with logic functions, the actions are similar to those performed during logic synthesis. The major difference lies in what must be done after the RTL description has been parsed. Whereas synthesis software is simply concerned with mapping an RTL description into a gate-level equivalent, using a standard cell library or some similar such target representation, and performing some minimizations along the way, BATG must understand the behavior of the RTL constructs that it encounters. It

must calculate input assignments and apply them to a network of RTL functions in order to coerce behaviors capable of exposing manufacturing defects or, alternatively, traverse the circuit in order to assist a designer in exercising and confirming the correctness of its behavior.

A starting point for BATG, when manipulating a circuit description, is to assign a set of values to a function, equivalent to the gate-level PDCF. For an AND gate, an input n-tuple is assigned such that a stuck-at fault on a single input causes the output of that AND gate to be functionally dependent on the presence or absence of the fault. An equivalent assignment at the RTL level—we shall refer to it as the primitive function test pattern (PFTP)—might be determined by studying the effects of stuck-at faults on the gate-level equivalent of the RTL function. This would lead to the development of a PFTP capable of detecting the fault. (See Section 7.8 for a discussion of behavioral fault modeling.) An alternative is to determine PFTPs behaviorally. For example, when loading a register, the PFTP could contain alternating 1s and 0s capable of detecting both stuck-at faults on inputs and shorts between adjacent inputs to the register. PFTPs could be designed to detect pattern sensitivity within a device. Error modeling (Section 12.4.2) suggests some PFTPs that can be useful for both fault detection and design verification.

A major difference between the PDCF and the PFTP is the fact that the PFTP could be a sequence of several vectors. Once the test has been defined, it must be justified, just as the assignment for a gate-level construct must be justified. This involves tracing back through connectivity to find elements driving the function. Propagation may or may not be necessary, depending on whether the user is concerned with performing design verification or test vector generation.

Data representation often shapes the response to an event or situation. For a gate-level ATPG the elements are basically simple logic devices, and except for the latches and flip-flops, there is usually no distinction between the inputs. Testability analysis tools, such as SCOAP, can help to differentiate between the inputs by identifying those that are easiest to control or observe, but otherwise the inputs have the same functionality and the data structures used to represent them can be rather elementary.

Data structures for behavioral level ATPG must support the LPM or similar such functions. If LPM functions are chosen as the prototypes, there will be 25 distinct data structures for the 25 functions. The data structures must contain the I/O connections and parametric information for the LPM, but there must also be entries for the PFTPs, there must be propagation knowledge similar to D-cubes, and there must be justification values. However, each of these will be much more complex. For example, if the value n is required from a counter, there will usually be many ways to obtain that value. It may be loadable, or it may be necessary to reset the counter and count up to n, or it may be possible to count up (or down) from the value currently present in the counter.

Whereas the gate-level ATPG has rather simple processing capability for the primitives that it recognizes, BATG subsumes these ATPG capabilities, but requires more complex processing capability as well. Some functions are purely combinational, while others are sequential. Sequential circuits may be quite elementary, such

as n-wide registers with parallel load capability. Others, including counters and state machines, represent behaviors that extend over a potentially infinite number of clock periods. Here, again, there is a wide range in degree of complexity. A counter can often be characterized by a simple expression, whereas in a state machine each state may have several alternatives for a next-state transition, requiring a considerably more complex case statement to represent its behavior.

State machine behavior is further compounded by the fact that there may be multiple interlocked state machines. Behavior of a state machine may be subordinated to the behavior of another state machine, at least in some of its states, or it may be subordinated to a counter, such that the state machine remains in state S_i until the counter reached some designated value.

12.6.4 Some Basic Behavioral Processing Algorithms

Interest in functional or behavioral level modeling for automatic test pattern generation reaches back more than two decades.[14] Functional modeling techniques for test pattern generation or fault propagation, while analogous to gate-level methods, must of necessity be more flexible. An algorithm for an RTL circuit can be partitioned so as to be expressed in terms of its data inputs and its control inputs, using Figure 12.4 as a paradigm. The control inputs select the input port(s), the operation to be performed, and possibly a destination.

To illustrate this, assume that we are to develop processing routines for an ALU. It may have several operations, including fixed-point addition and subtraction, AND, OR, invert, complement, all 0s, and all 1s. In addition, it may be able to pass an argument straight through from an input port to the output port without being altered. Each of these operations requires a specific setting on the control lines. During justification, if an all-0s or all-1s vector is required on the output, the appropriate op-code is established on the control lines and the input ports are ignored. If a specific value is required on an output port and if the control lines can be set to pass a value from an input port directly to the output, then that setting is used for justification. If a pass-through capability exists for two input ports, then a decision may have to be made as to which port should be chosen. If a straight pass-through does not exist, then a logic operation can be selected. The desired value can be placed on one port while the all-0s (all-1s) vector is placed on the other port, and an OR (AND) operation is selected.

The entire process just described can be structured as a sequence of IF...THEN statements. The possibility exists that one or more operations available in a function may be blocked by virtue of the fact that the circuit, as designed, does not implement the operation. The all-0s operation may exist in the ALU but the op-code is not implemented. If the BATG discovers that the operation is unavailable, it is marked as BLOCKED, so no attempt is made to use it again for justification or propagation operations. Note that this is not the same as a conflict. A BLOCKED operation occurs if the particular operation exists in the function but is not used in the design. A conflict occurs when, during backtracing, different paths require different values from the same source.

Another significant difference between gate-level and functional primitives lies in the fact that the propagation and justification rules for sequential devices can be, and usually will be, sequences of operations rather than single operations. As a rather simple example, the test for an edge-triggered flip-flop may be expressed as a sequence in which a 1 on the Data line is clocked in, and then the Clear line is exercised to confirm that it will, indeed, reset the flip-flop output to 0 (see Section 5.3.6 for a discussion of SPS, a sequential D-algorithm).

A more complex example of sequential devices is the serial/parallel shift register or the counter. Processing is complicated by the presence of a Hold state, during which the counter may be required to be inactive. The range of functional operations can be described symbolically as

L	left shift
R	right shift
P	parallel load
H	hold (do nothing)
U	count up
D	count down
C	clear

Any particular operation that must be performed can be expressed in terms of these operators.

Example A 1 can be justified on the i^{th} output of a counter with the sequence CH*{UH*}2^i. The notation indicates that a 1 is obtained by performing a clear, followed by zero or more hold operations (the asterisk denotes an arbitrary number of consecutive operations of the type specified by the operator to its immediate left). Then the entire sequence in braces, which is a count up followed by zero or more hold operations, is repeated the number of times indicated by the number following the right brace, in this case 2^i times. ■■

The abstract operations must relate to real counters, either those available from semiconductor manufacturers or those designed by the user. The operations correspond to I/O pins that perform the operation. It is also necessary to relate operations to such things as rising or falling clock edge, depending on which edge enables the activity.

Rules can also be defined for propagation and implication. Again, the rules are expressed functionally. The signal values are analogous to D-cubes in that they specify, for a D or $\overline{\text{D}}$ on an input, exactly what signal(s) must appear on other inputs to make the output sensitive to the D or $\overline{\text{D}}$. For a shift register that has a clear line K and control lines S_1 and S_0 which may select hold, parallel load, shift left, and shift right, Table 12.1 expresses some of the propagation rules. In this table, $y(i)$ represents the present value in the flip-flop at position i and $Y(i)$ represents the new value

TABLE 12.1 Propagation Rules

Composite	K	S_1	S_0	$y(i-1)$	$y(i)$	$y(i+1)$	A	$Y(i)$
C/H	D	—	—	—	1	—	—	$\overline{D}$
L/H	0	D	0	1	0	—	—	D
P/H	0	0	D	—	1	—	0	$\overline{D}$
L/L	0	1	0	D	—	—	—	D
H/H	0	0	0	—	D	—	—	D
R/R	0	1	1	—	—	D	—	D
P/P	0	0	1	—	—	—	D	D

after clocking the register. For entry *u/v* in the composite column, *u* denotes the action taken by the fault-free circuit and *v* denotes the action taken by the faulted circuit. The first line defines the conditions for propagating a D on the Clear line to output *i*. It requires first clocking a 1 into register flip-flop $y(i)$. The faulty circuit will perform a hold operation. Propagating a D through control line S_1 to output $y(i)$ requires a 1 in register bit position $y(i-1)$ and a 0 in position $y(i)$. Table 12.1 is not a complete list. For example, propagation through a control line could also be accomplished with a 0 in $y(i-1)$ and a 1 in $y(i)$.

Implication tables can also be expressed functionally. They can be created via composition; that is, if a D ($\overline{D}$) occurs on one or more lines, then the results can be computed individually by first setting D = 0 ($\overline{D}$ = 1) and performing the computation, then setting D = 1 ($\overline{D}$ = 0) and again performing the computation. After computing each case individually, set the output or internal state variable to 0, 1, or *x* if it has value 0, 1, or *x* for both good circuit and faulty circuit. If it assumes value 0 (1) for good circuit and 1 (0) for faulty circuit, set it to D ($\overline{D}$). If it assumes value *x* for good circuit, set it to *x*. If it assumes *x* only for the faulty circuit, then its value depends on whether the user wants to consider possible detects or only absolute detects.

The following paragraphs describe a functional test pattern generation algebra developed for use in conjunction with HDLs.[15] First, define $U = \{0, 1, D, \overline{D}\}$. Then, if S_i is a subset of U, x^{S_i} denotes the fact that $x \in S_i$, and the following equations hold:

$$x^{S_i} \cdot x^{S_j} = x^{S_i \cap S_j} \tag{12.1}$$

$$x^{S_i} + x^{S_j} = x^{S_i \cup S_j} \tag{12.2}$$

$$x^{S_i} + x^{\bar{S}_i} \cdot y = x^{S_i} + y \tag{12.3}$$

$$x^{S_i} + x^{S_i} \cdot y = x^{S_i} \tag{12.4}$$

For the AND operation, the following equations define all of the combinations on the inputs *a* and *b* which will produce the indicated value on the output *c*:

The Invert function is obtained by complementing the superscript; that is, if $b = \overline{a}$, then $b^i = a^{\overline{i}}$, where $\overline{S}_i$ is obtained by complementing each of the individual elements contained in S_i.

$$c^0 = (ab)^0 = a^0 + b^0 + a^D b^{\overline{D}} + a^{\overline{D}} b^D \tag{12.5}$$

$$c^1 = (ab)^1 = a^1 b^1 \tag{12.6}$$

$$c^D = (ab)^D = a^D b^1 + a^1 b^D + a^D b^D \tag{12.7}$$

$$c^{\overline{D}} = (ab)^{\overline{D}} = a^{\overline{D}} b^1 + a^1 b^{\overline{D}} + a^{\overline{D}} b^{\overline{D}} \tag{12.8}$$

The equations for the OR gate can be computed from the equations for the AND gate and the inverter. Equation (12.5) states that if c is an AND gate with inputs a and b, then a 0 is obtained on the output by setting either a or b to 0 or by putting D and $\overline{\text{D}}$ on the inputs. Note from Eq. (12.8) that a $\overline{\text{D}}$ on both inputs does not put a 0 on the output but, rather, a $\overline{\text{D}}$.

These basic equations for the logic gates can be used, together with the four rules, to compute D cubes for more complex functions.

Example JK flip-flop behavior can be represented by

$$Q = J\overline{q} + \overline{K}q$$

where q is present state and Q is next state. D-cubes are computed using

$$\begin{aligned} Q^{\text{D}} &= (J\overline{q} + \overline{K}q)^{\text{D}} \\ &= (J\overline{q})^0(\overline{K}q)^{\text{D}} + (J\overline{q})^{\text{D}}(\overline{K}q)^0 + (J\overline{q})^{\text{D}}(\overline{K}q)^{\text{D}} \\ &= J^0K^0q^{\text{D}} + J^{\text{D}}K^0q^{\text{D}} + K^{\overline{\text{D}}}q^1 + J^1K^{\overline{\text{D}}}q^{\text{D}} + J^{\text{D}}K^{\overline{\text{D}}}q^{\text{D}} \\ &\quad + J^1K^1q^{\overline{\text{D}}} + J^{\text{D}}K^1q^{\overline{\text{D}}} + J^{\text{D}}q^{\overline{\text{D}}} + J^1K^{\overline{\text{D}}}q^{\overline{\text{D}}} + J^{\text{D}}K^{\overline{\text{D}}}q^{\overline{\text{D}}} \end{aligned}$$

The result can be used to create a table of propagation and justification cubes for both D and $\overline{\text{D}}$ values. ■■

The basic operators can also be used to create tables for more complex functions. The adder can be created one bit position at a time. The sum and carry tables are created from the exclusive-OR and the AND, respectively. These are then used to build up one complete stage of a full adder which is then used to build an n-stage adder. The multiplexer can be expressed in equation form as

$$F = S \cdot A + \overline{S} \cdot B;$$

where S is the select line, A and B are inputs, and F is the output. Since it is now expressed in terms of OR and AND gates, the cubes for the equation can be generated.

12.7 THE SEQUENTIAL CIRCUIT TEST SEARCH SYSTEM (SCIRTSS)

SCIRTSS was a research system that evolved over a period of several years in the 1970s and 1980s at the University of Arizona. Its purpose was to investigate the use of RTL constructs in behavioral ATPG. The RTL language used for this purpose was AHPL (a hardware programming language). Several novel concepts resulting from this research will be described here.

12.7.1 A State Traversal Problem

ATPG problems caused by sequential circuits were discussed in Chapter 5. The additional time dimension introduced by asynchronous circuits, including such things as pulse generators, is far beyond the comprehension of ATPG programs. However, even when a circuit is completely synchronous, complexity issues are still capable of thwarting the ATPG. Consider the state machine implemented in Figure 8.44. A fault on input 3 of gate 23 requires sensitization values 1, 0, 0 on flip-flops Q_2, Q_1, Q_0. If the ATPG performs a reset on the circuit, it would appear that the problem is rather easily solved by driving Q_2 to a 1, seemingly an easy solution.

However, from the state transition graph for that circuit, Figure 12.5, it can be seen that it is necessary to pass through state S_3 to get to state S_4. But state S_3 corresponds to Q_2, Q_1, Q_0 = 0, 1, 1. In other words, after performing a reset, the ATPG must be clever enough to get Q_2, Q_1, Q_0 = 1, 0, 0 during backtrace by first driving the flip-flops to Q_2, Q_1, Q_0 = 0, 1, 1. But it is not in the nature of a gate-level ATPG to try to put a 1 on the output of a flip-flop by driving it to the 0 state. The gate-level ATPG can deal with this situation only if it is given knowledge about the nature of the function, or if it thrashes about randomly and fortuitously stumbles upon a solution.

The above-mentioned problem occurs because the typical gate-level ATPG does not take a global view of a circuit. It sees the logic gates but not the state machine. Rules exist for the primitives (PDCFs, propagation D-cubes, etc.), and the ATPG processes these primitives in isolation. There are relationships between flip-flops in the circuit that are not obvious at the gate level. However, from a graph it is often a trivial exercise to determine how to get from the reset state, S_0, to the objective state

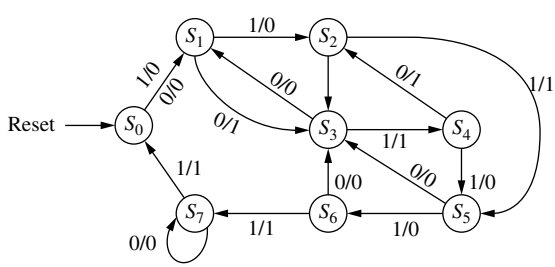


Figure 12.5 State transistion graph.

S_4. This observation is the basis for SCIRTSS (Sequential CIRcuit Test Search System).[16,17] SCIRTSS uses two models of a circuit: a detailed gate-level circuit description and an HDL description expressed in AHPL (a hardware programming language).

SCIRTSS employs a D-algorithm to find a sensitization state for a selected fault. The sensitization state is a set of binary values that, when assigned to flip-flops, latches, and primary inputs, causes a sensitized path to extend from the fault source to either a primary output or to a stored state device, which may be a latch or flip-flop. When the fault propagates to a stored state device, it is said to be *trapped* in that element.

The D-algorithm is strictly combinational; it does not attempt to create multiple time images in order to propagate faults through sequential elements. Once the sensitization state has been computed, the D-algorithm is done. The operation up to this point is essentially the same as that performed by a scan-based test.

In the next step, SCIRTSS diverges from the scan approach. Scan is essentially done at this point, it simply remains to scan in the sensitization state, apply a functional clock, and then shift the captured data to the scan-out pin (cf. Chapter 8). SCIRTSS, however, attempts to drive the circuit from its present state to the state that sensitizes the fault. This is accomplished through the use of the RTL description. To do this, SCIRTSS enters the *sensitization search* phase where it employs an AHPL description of the circuit. The AHPL description may specify a transition directly to a single next state or it may identify several reachable next states, as well as the conditions that determine which of the next states is selected. The sensitization search is essentially a tree search in which SCIRTSS, starting at the present state, or possibly the reset state, tries to find a sequence of inputs that drive the circuit to the sensitization state.

If the sensitization search is successful, then a sequence of inputs has been found which, starting at the present state, either makes the fault visible at an output or causes the fault to become trapped in a latch or flip-flop. If the fault becomes trapped, then SCIRTSS enters the *propagation search*. In this phase, SCIRTSS attempts to drive the circuit through a sequence of states that cause the fault to become visible at an output. This phase, like the sensitization phase, tries to control the behavior of the circuit by using the AHPL description to compute transitions.

When a complete test has been achieved, including both sensitization and propagation sequences, SCIRTSS again resorts to the gate-level description. This time, the gate-level description is used to perform fault simulation. Fault simulation has three objectives:

- It must confirm that the fault was detected.
- It must identify any other faults that were detected.
- It must identify any faults that became trapped by the applied sequence.

If there are trapped faults, then one of them is selected for processing and SCIRTSS again goes into propagation search. If there are no trapped faults, then SCIRTSS goes back to sensitization search. The entire process is illustrated in Figure 12.6.

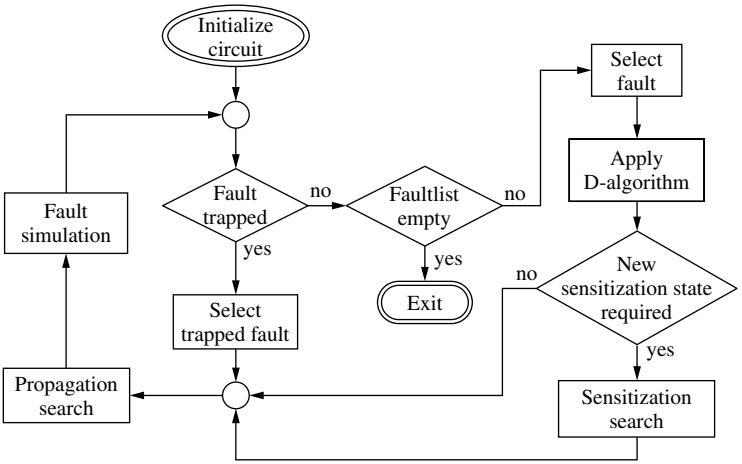


Figure 12.6 SCIRTSS flowchart.

The tree search conducted by SCIRTSS is subject to combinatorial explosion. With m inputs, a search depth of k states could produce a tree with as many as 2^{mk} sequences, resulting in a need for massive amounts of memory and CPU time. One way to reduce the search space is to view the control section and data path of a circuit as distinct entities (cf. Figure 3.1). When data are altered in registers that belong to the data path, these events are viewed strictly as data transfers, not as state changes. Note that an allowance must be made for ALU operations that affect status registers which, in turn, affect state transitions.

Since this is essentially a search problem, and the field of artificial intelligence (AI) has been refining state-space search algorithms for several decades, it made sense to turn to the field of AI and borrow some of the techniques developed there. Two basic tenets of AI that were applied to SCIRTSS were as follows:

1. A limited n-level search may have a greater payback than an exhaustive $n - 1$ level search.
2. Self-modifying methods, based on previous results, can improve the probability of pursuing the correct path in a search.

As a part of this strategy, when searching for sequences of state changes, heuristics were employed. A *heuristic* is anything (in this case a number) that guides a search, or otherwise helps to discover a solution. However, the heuristic is not capable of proof. The heuristics assign a value to each circuit node during a search according to the following formula:

$$H_n = G_n + w \cdot F_n$$

In this formula G_n is the distance from the starting node to node n, F_n is a function of any information available about node n as defined by the user, and w is a constant that determines the extent to which the search is to be directed by F_n. The object is to find the easiest or shortest path to an objective state.

Example The state transition graph in Figure 12.5 is used to illustrate a sensitization strategy. The gate-level model for this circuit is given in Figure 8.44. If you did Problem 8.10(b), you may find it interesting to compare your Verilog model to the graph in Figure 12.5. A stuck-at-1 on the input to gate 23 driven by the inverter labeled gate 1 is chosen as the target fault. The combinational D-algorithm determines that a path can be sensitized from the fault to the output if the circuit is in state $Q_2, Q_1, Q_0 = (1, 0, 0)$. Therefore, a sequence of inputs is needed that cause the circuit to transition to state S_4. An assumption is made that the current state of the circuit is indeterminate.

From the Verilog description it can be determined that it is only possible to reach state S_4 from state S_3. It is possible to reach state S_3 from four states, as indicated from the search tree in Figure 12.7. Three of these states, S_1, S_2, and S_5, can themselves be reached from two states, while state S_6 can only be reached from S_5. State S_1 can be reached from S_0, which can be reached by applying a Reset to the circuit.

A complete sequence for sensitizing the selected fault consists of applying a 0 to the Reset, releasing it, then applying the sequence IN = {X, 0, 1}. Although the first value is listed as a don't care, it must nevertheless be a 0 or 1. As each of these is applied, the circuit must be clocked. Then, after reaching state S_4, the stuck-at input must have value 0, requiring that IN = 1.

Finally, the entire sequence is simulated at the gate level to confirm its effectiveness and to determine whether other faults are detected. Note that when creating a tree, multiple occurrences of states appear. For instance, S_1 is a leaf node. It may be productive to pursue the path $\{S_0, S_1, S_2, S_3, S_4\}$ for the reason that within the context of a larger circuit, this path may be easier to set up. Another justification for the longer path may be to exercise an arc that had not previously been exercised. In such a case, weighting schemes may be counterproductive. Also note that it can be seen from the tree that sometimes it is possible to reach the objective state with a shorter sequence, such as if the circuit is currently in state S_5. ■■

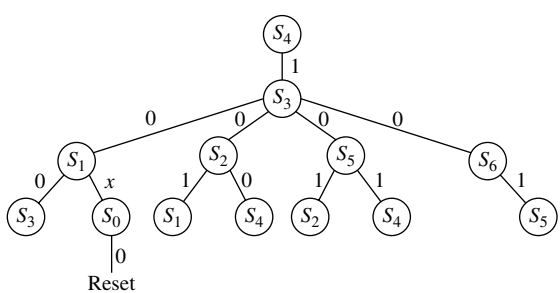


Figure 12.7 Sensitization search tree.

At the conclusion of fault simulation, one or more faults may be trapped in the flip-flops. If the output of gate 16 is S-A-1, it will not be detected when the reset is applied, nor will the first two transitions to S_1 and S_3 distinguish between the good circuit and the faulty circuit. However, in the transition to S_4, the faulted circuit goes to S_5; hence a $\overline{D}$ is trapped in Q_0. From the state graph, Figure 12.5, it is seen that IN = 0 causes an output of 1 from the good circuit and an output of 0 from the faulted circuit. Furthermore, it was not even necessary to clock the circuit.

If no faults are trapped when in state S_4, and if the output of gate 12 S-A-0 is selected from the fault list, the D-algorithm would start by assigning a PDCF of (1, 1) to the inputs of gate 12. The fault would propagate to Q_0 and become trapped if IN = 0 and Q_2, Q_1, Q_0 = 0, 0, 1, and the circuit is clocked. From the graph, it is seen that there are a number of ways that the sensitization search can go from S_4 to S_1. The signal IN can be set to 0 or 1, but it is also possible to reset the circuit and go immediately to state S_0; hence there are three possible successor states to S_4. Furthermore, the transition from S_0 to S_1 is trivial to compute.

However, it may be preferable to force the circuit through states $S_5 \rightarrow S_6 \rightarrow S_7$ from state S_4 in order to exercise additional logic and perhaps detect faults that might otherwise require individual processing. This can be done with the heuristic. The w term and the F_n term in the heuristic can be chosen to force SCIRTSS to go through those additional states rather than transition directly to S_1. It may also be desirable to modify the heuristics after the process has run for some time in order to force state transitions through other logic. This modification on-the-fly requires that intermediate results be available for inspection.

The trapped fault in Q_0 can be processed by the D-algorithm or it can be processed directly from the graph. The D-algorithm can propagate the D in Q_0 to OUT (through gate 22) by setting IN = 0. The value IN = 0 could also have been determined from the graph. The fault-free circuit is in state S_1 and the faulted circuit is in state S_0. Therefore, it is easily determined from the graph that IN = 0 causes different outputs from the two states.

It was mentioned that SCIRTSS employs two models, a gate-level model and an AHPL model. The AHPL model permits circuit exploration at a level of abstraction that avoids many of the pitfalls of gate-level ATPG. Meanwhile, the gate-level model can be quite flexible, including timing and transistor level primitives in order to uncover serious timing problems with vectors developed by SCIRTSS. The only restrictions on the gate-level model are those imposed by the gate-level ATPG used to sensitize faults.

Some observations concerning SCIRTSS:

1. It must be possible to correlate abstract states S_i with values on the flip-flops; for example, if state S_4 corresponds to the assignment Q_2, Q_1, Q_0 = (1, 0, 0), then SCIRTSS must know that.
2. The heuristics can be updated to reflect successful state transition sequences. In the example given, a transition from S_4 to S_6 or S_7 is performed more quickly if the first transition is directly to state S_5 rather than to S_2.

3. It is possible to give up on a fault and succeed later when sensitization search starts from another state.
4. It is not necessary to have a completely specified objective state. If the D-algorithm leaves one or more flip-flops in the state machine unassigned, then the objective may be a set of states determined by setting the Xs to 1 and 0. A sensitization search is successful if any state in the set is reached.
5. During gate-level simulation it is necessary to keep track of fault effects of all faults of interest since a fault may, over time, affect both data registers and one or more control flip-flops. This could cause the fault to mask its own symptoms.
6. Arguments required in the data path to cause a propagation may originate in other registers; therefore it may be necessary to derive sensitization sequences in which an argument is first loaded from a data port into a register or accumulator and then used to propagate the fault forward to an output or flip-flop.
7. SCIRTSS, as described, frequently employed user-suggested trial vectors at the data ports. These included such typical vectors as the all 1s, the all 0s, the sliding diagonal (cf. Section 10.3), and so on. In addition to stuck-at faults, these vectors also detected shorts between adjacent pins, as well as problems caused by excessive numbers of pins switching simultaneously.

12.7.2 The Petri Net

State transition graphs are somewhat limited in their ability to model activities that take place in digital circuits. In many circuits it is necessary for two or more processes to occur before a subsequent task that is dependent on the results of these earlier tasks can proceed. Each of these preceding tasks may execute simultaneously, or they may execute serially. The order is usually immaterial. In a typical state machine configuration registers, status registers and mode control registers may all need to be configured to some particular value before another task can proceed. Some of these are loaded by software, while some of these registers are loaded by hardware as a result of other functions executing in the hardware.

Some hardware design languages provide constructs to accommodate these asynchronous or independent activities. Typical among these are such constructs as FORK, which causes several events to run concurrently, and JOIN, which specifies that a task cannot proceed until those events spawned by the FORK have all completed.

The Petri net is a useful mechanism for describing the necessary convergence of events that must occur in order to trigger a subsequent event. The *Petri net* is a bipartite, directed graph $N = \{T, P, A\}$ where[18]

$T = \{t_1, t_2, ..., t_n\}$ is a set of *transitions.*

$P = \{p_1, p_2, ..., p_m\}$ is a set of *places.*

($T \cup P$ form the nodes of N.)

$A \subseteq \{T \times P\} \cup \{P \times T\}$ is a set of directed arcs.

A *marking* of a Petri net is a mapping:

$$M : P \rightarrow I$$

where $I = \{0,1,2, \ldots\}$. M assigns tokens to places in the Petri net. M can also be thought of as a vector whose ith component represents the number of tokens assigned to place p_i. A Petri net in which every transition has exactly one input place and one output place is a state machine.

A place may have a token (sometimes called a marker) or it may be empty. If all of the input places to a transition have tokens, then the transition is *enabled*, and this permits the transition to fire. In the process of firing, the transition moves one token from each input place and puts one token into each output place.

Figure 12.8 illustrates a Petri net used to represent flow of control in a program.[19] The transitions, represented by bars, can only be connected to places, represented by circles, and the places can only be connected to transitions. The places from which arcs emanate are called input places of a transition, and the places on which an arc terminates are called output places of a transition.

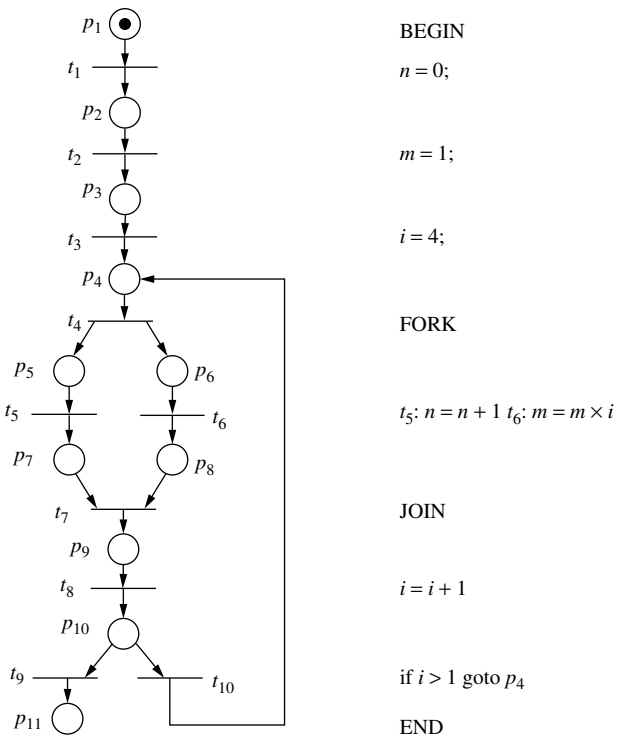


Figure 12.8 Program described by Petri net.

The place designated p_1 has a token. Since all of the input places to t_1 have markers, it is enabled and fires. Upon firing, the token is transferred to p_2. When transition t_2 fires, a token is placed in p_3. At transition t_4 a token is deposited in both p_5 and p_6. Transition t_7 will not fire until both p_7 and p_8 have tokens. An important point to note is that place p_{10} has a single token, and place p_{10} is connected, via input arcs, to transitions t_9 and t_{10}. Since there is only a single token in p_{10}, both t_9 and t_{10} are enabled but only one of them can fire. In this case, t_9 and t_{10} are said to be in conflict. A conflict occurs when two transitions share a place and both become enabled, but there is a single token. With a single token in p_{10}, only one of t_9 and t_{10} will fire, and the first one to fire will disable the other. It is possible for a place to have more than one token simultaneously, in which case it is said to be *safe*. If at any time during operation of the net, no transition is ruled out as a transition that may fire some time in the future, the net is said to be *live*.

A Petri net can represent a hardware implementation as well as a computer program. In fact, although our interest is in using Petri nets to represent hardware behavior, they have been used to represent many different processes, including chemical processes where input places represent reacting chemicals, transitions represent reactions, output places represent the results of a reaction, and tokens represent the number of molecules of a given type.

The Petri net has been used in conjunction with SCIRTSS.[20] It was used to reduce the search cost required to reach a goal state and also to generate input vectors used to expand the state space nodes. Because even synchronous designs frequently require that several events be properly set up before a subsequent operation can occur, the Petri net can sometimes provide more insight than a state machine representation when trying to describe complex circuit behavior.

The following circuit will be used to illustrate the use of the Petri net, as well as to illustrate some tree search techniques used to find a solution for a given set of goals.[21] The original circuit was expressed in AHPL (a hardware programming language),[22] This example has been translated to Verilog.

```
module sp(reset, clk, inp, mor);
input reset, clk;
input [3:0] inp;
output [3:0] mor;
reg [3:0] mor, mdr, ac;
reg [2:0] ir, state;
always @ (posedge clk or negedge reset)
  if (!reset)
   state = 3'b000;
  else
   case (state)
    3'b000: state = 3'b001;
    3'b001: begin
```

```
            ir = inp[2:0];
            if (inp[3] == 0) state = 3'b010;
          end
    3'b010: if (ir[0] == 1) state = 3'b011; // -> state 3
          else state = 3'b111; // -> state 7
    3'b011: case (ir[2:1])
            2'b00: state = 3'b100;
            2'b01: state = 3'b101;
            2'b10: state = 3'b110;
            2'b11: state = 3'b001;
          endcase
    3'b100: begin
            mdr = inp;mor = ac;
            state = 3'b001;
          end
    3'b101: begin
            ac = inp; mor = 4'bzzzz;
            state = 3'b001;
          end
    3'b110: begin
            ac = ac & mdr; mor = ac;
            state = 3'b001;
          end
    3'b111: begin
            ac = ac >> 1;  mor = ac;
            state = 3'b001;
          end
  endcase
endmodule
```

In this example the case statement represents a state machine. In state 3 (3′b011) there is another case statement. This case statement represents a multiplexer, the next state assignment depends on the values of bits 1 and 2 of the instruction register. A synthesis program distinguishes between the two by noting that the outer case statement is controlled by a clock edge.

We will not present a structural model of this circuit, but we can, nevertheless, postulate the existence of a fault that becomes sensitized—that is, whose PDCF (cf. Section 4.3.2) is satisfied—when the circuit is in state {ir, mdr, ac} = {3′b101, 4′b11XX, 4′b11XX} or in state {ir,mdr,ac} = {3′bXX0, 4′bXXXX, 3′b1XX}. The goal tree for the initial conditions corresponding to these sensitization requirements is shown in Figure 12.9.

The goal labeled P_0 is the output place of two transitions, t_1 and t_2, which correspond to the two sensitization states for the fault. This represents an OR condition: If

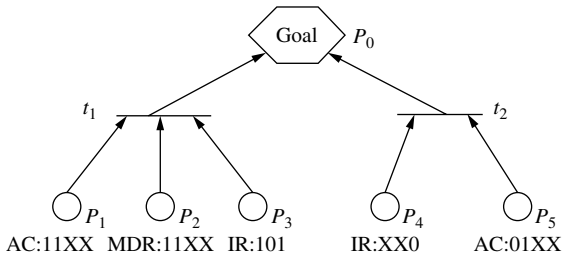


Figure 12.9 Initial goals for search.

either transition t_1 or t_2 fires, then a token will be deposited in P_0 and the goal is satisfied. Note, however, that by virtue of the rules for a Petri net, t_1 cannot fire unless there are tokens in P_1 AND P_2 AND P_3, while t_2 cannot fire unless there are tokens in P_4 AND P_5. The actions represented by places P_1 through P_5 are listed underneath them in the figure.

The diagram in Figure 12.10, at this point, represents the initial sensitization conditions for the circuit. What we hope to achieve is the creation of an input sequence that will drive the circuit into one or the other of the two transitions depicted in Figure 12.9. Consider place P_1. What must be done to get bits 3 and 2 of register ac set to 1? A search of the Verilog description reveals that ac is loaded from the input port when the circuit is in state 5. So, if state = 5 and inp = 4′b11XX, then in the next clock period ac = 4′b11XX. But the requirements can also be satisfied when in state 6. Observe that in state 6 ac receives the AND of ac and mdr. So, if ac = 4′b11XX AND if mdr = 4′b11XX, on the next clock ac will receive (actually, retain) the value 4′b11XX.

A complete second stage of the Petri net is given in Figure 12.10. This is not a complete tree; several more stages are required to reach leaf nodes for this graph. We leave it as an exercise for the reader to identify the places and complete the graph.

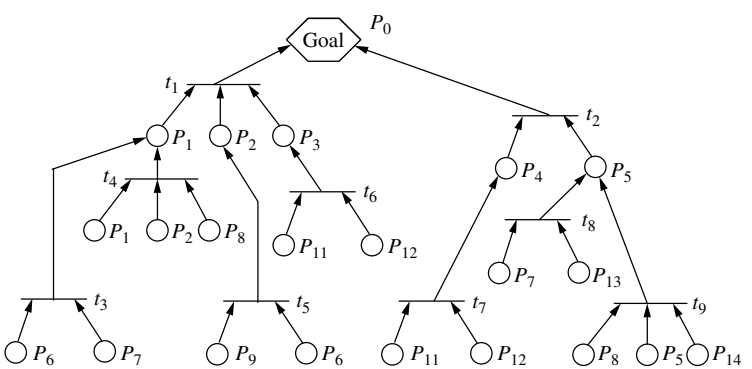


Figure 12.10 Second level of Petri net goal tree.

The objective is to develop one or more paths that define a transition either from the current state of the circuit or from a reset state and, by means of the Petri net, identify a sequence of inputs that will drive the circuit to the Goal. In either case, the traversal is directed by means of input stimuli. By virtue of having several paths from leaf nodes to the Goal, several options exist. One option is to traverse the shortest path from the leaf node to the Goal. Another option is to traverse a path that includes branches that have not yet been traversed in order to exercise heretofore unexercised logic.

It may be desirable to spread out the traversals, choosing different paths each time, so that a manufacturing test program exercises all paths approximately the same number of times, rather than exercise the same path repeatedly. However, recognizing that tester time can be quite expensive, the goal of a manufacturing test program usually is to be as short and efficient as possible, so it may be desirable to find the "least cost" path. From Figure 12.10 it can be seen that the searches can grow out of control quickly, and the example circuit was rather small. The SCIRTSS project spent much effort developing cost functions to help navigate through the logic and prune the search trees in order to find shortest paths as well as to control the growth of goal trees.

12.8 THE TEST DESIGN EXPERT

The Test Design Expert (TDX) was a commercial endeavor motivated by the SCIRTSS system, and it bore some resemblence to it. But TDX included strategies, techniques, and refinements that took it beyond SCIRTSS, and some of the features that it had in common with SCIRTSS were evolved and refined. Inputs to TDX included a netlist and an RTL description. It also employed a map file to link storage elements in the RTL with their instantiated counterparts in the netlist.

12.8.1 An Overview of TDX

Like SCIRTSS, TDX used search heuristics to explore RTL models, but as it evolved it added additional software tools. The Supervisor was continuously selecting and applying tools to the task of generating vector sequences. TDX included a gate-level ATPG called DEPOT (DEductive, Path-Oriented Trace), which implemented both the D-Algorithm and the PODEM algorithm, and TDX could select either of them under Supervisor control.

TDX included a testability analyzer that was a variant of SCOAP, and the C/O numbers that it generated from the netlist were used by DEPOT to help find the best path through combinational logic. The best path usually meant propagating a fault to a destination storage element or primary output while making the smallest possible number of justification assignments to storage elements. This was an important consideration because as more logic assignments are made to flip-flops while sensitizing a fault in a sequential circuit, the more difficult it becomes to justify the values on those flip-flops while trying to sensitize the fault or propagate the

fault effect through the RTL code. Behavioral C/O numbers were also computed and used.

A full-timing, gate-level concurrent fault simulator was integrated with the other components of TDX. It provided a detailed analysis of fault coverage for the vectors generated by TDX. It also performed logic simulation to verify that sequences generated by BATG had the intended effect and were not side-tracked by races and hazards. Outputs from TDX included a test vector file, a response file, and various reports, including fault coverage and testability analysis information.

The key components of TDX are shown in Figure 12.11. Input files included a netlist, an RTL description and a map file, which linked gate-level flip-flops and latches with their RTL counterparts. The RTL description could be provided in VHDL or Verilog. Regardless of which language was used to describe the circuit, when parsed, it was translated into a behavioral intermediate form (BIF) that expressed the RTL code as a collection of cause and effect rules. The netlist could also be provided in either VHDL or Verilog. A faultlist compiler read the netlist and produced a list of the traditional stuck-at faults. The search heuristics were a collection of strategies, algorithms, and heuristics that could be invoked as needed by the TDX Supervisor to solve problems.

The Supervisor controlled the overall operation of TDX. The first step was to read in the netlist and the BIF and compile a knowledge base. The knowledge base differs from a data base in that it "is more explicit about the objects in its universe and how information flows between them."[23] The Supervisor analyzes the RTL description and breaks it down into groups that correspond to state machines, counters, multiplexers, and other familiar structures. Some parts of a circuit are described using long series of detailed RTL equations. Those parts of the design are stored as equations.

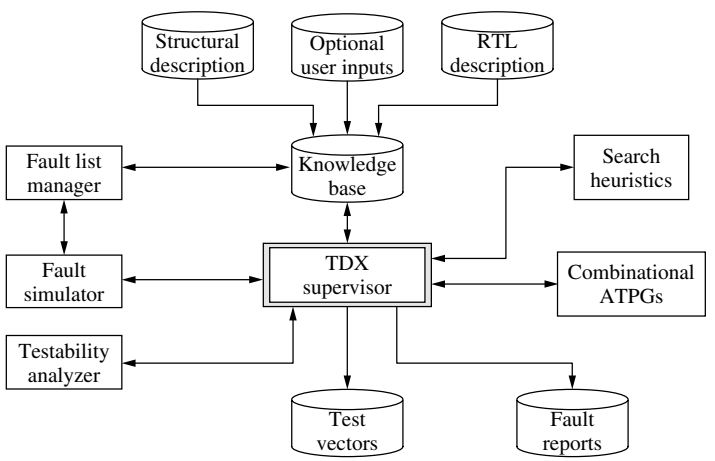


Figure 12.11 The Test Design Expert (TDX).

When reading in the RTL description, much of the initial processing was similar to that performed by a synthesis program; that is, RTL constructs are recognized and mapped into appropriate data structures corresponding to the common hardware functions. In one respect, however, TDX digressed from synthesis programs. The synthesis program works just fine with gate-level modules intermingled with RTL. In fact, a synthesis program might accept either a pure RTL, or a mixed RTL/gate description and produce identical gate-level netlists from them. The synthesizable subset handles low-level detail quite well, but may have trouble with higher levels of abstraction. TDX, conversely, was quite adept at handling higher levels of abstraction, but often stumbled with circuits that contained too much low-level detail. Complex, handcrafted modules in the data path part of the RTL that obscured functionality could sometimes prove difficult for TDX.

When the Supervisor invoked the fault list compiler to compile a fault list, it would compile either a full fault list or a fault sample of a size chosen by the user. The fault simulator was a full-timing, gate-level concurrent fault simulator that could accurately fault simulate both synchronous and asynchronous circuits. It was tightly integrated with the rest of the TDX system, so it could fault simulate a sequence of arbitrary length, pass control back to the Supervisor which would then invoke the ATPG, and then it could regain control from the Supervisor and resume fault simulating from the point where it previously left off. Alternatively, it could operate standalone on the same fault list that had been previously processed by TDX.

TDX could initialize a circuit or it could accept initialization stimuli from the user. Often, particularly when the circuit required complex, timing-critical sequences, the designer could accomplish the initialization more efficiently. Sometimes it was preferable to use design verification suites to get coverage up to a certain level. In such cases it was not necessary for TDX to generate vectors until the productivity of the user's vectors began to diminish.

When the user provided test vectors, these would be passed directly to the fault simulator, which would determine the coverage for these vectors. The faults that were detected by the test vector suite would be dropped by the fault list manager, so there would be no effort to generate tests for those faults. There were several hooks included in the fault simulator to permit it to communicate with the Supervisor. The Supervisor could, at any point during the process of test generation, query the fault simulator and determine which faults had been detected and which undetected faults had produced error signals that caused one or more flip-flops to assume an incorrect value.

Some of the capabilities of TDX included:

Gate-level combinational ATPG (D-algorithm and PODEM)

Trapped fault propagation

Controllability/observability analysis (gate level and behavioral)

Creation/manipulation of goal trees

Constraint propagation

Functional walk

Learn mode

Like SCIRTSS, TDX could target undetected faults for sensitization or it could identify trapped faults. If the Supervisor determined that there were trapped faults, it could choose one for propagation. If several trapped faults exist, the Supervisor could select one based on various criteria. The criteria were not hard and fast, they could vary during a run. As a result, a fault trapped in a flip-flop might be selected at one point during test generation, and the same fault trapped in the same flip-flop might be rejected at some other time during a run.

For example, suppose a fault becomes trapped in a flip-flop. Suppose that flip-flop becomes immediately observable at an output if a tri-state enable is set to 1, and suppose the tri-state enable is easily controllable. It is possible that several undetected faults are trapped in that flip-flop. In that case, it is desirable to enable the tri-state control and detect the faults. However, it is also possible that another flip-flop has trapped faults, and those faults originate in a region of the design where controllability and observability are very difficult. The other flip-flop, even if it requires a more complex sequence to flush out the faults, may be a more desirable objective.

The concept of targeting trapped faults was discussed in Section 7.9.2, when the SOFTG system was discussed. There it was pointed out that the tendency to grab a trapped fault must be tempered by the realization that a trapped fault can lead to a dead end. This is illustrated in the circuit of Figure 12.12, a variable-length byte-wide shift register. The shift length is programmed by loading a value in register R_S that determines which of the registers R_1–R_{16} will be selected and clocked into destination register R_{17} on the next active clock edge.

Suppose a fault effect appears in register R_3. It may be an opportune time to propagate that fault forward and position it closer to an output. But if R_3 is not currently selected by the multiplexer, then it must be selected by loading the correct value into R_S through select bits $S_{3\text{-}0}$. However, note what happens when the value at $S_{3\text{-}0}$ is clocked into R_S. The contents of R_3 are propagated to R_4 and are replaced by the contents of R_2. A knowledgeable human would recognize and allow for that possibility by loading R_S with the bits required to choose R_4.

The problem of dead ends in sequential logic can be quite serious. The problem occurs regardless of whether the fault became trapped serendipitously while another fault was the object of propagation, or the fault may be one that was sensitized by DEPOT. In other words, the process of selecting and sensitizing a fault may succeed in its effort to propagate a fault effect from the fault origin to a target flip-flop, but it may, in the sensitization process, produce a trapped fault that cannot be further propagated.

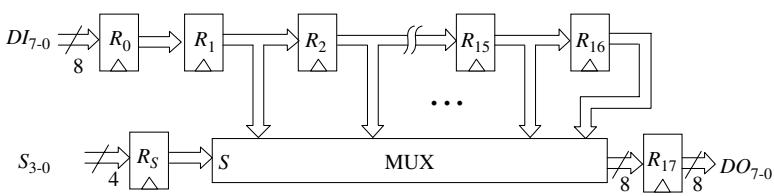


Figure 12.12 Dead end for a trapped fault.

Note that a strategy that might be employed by an experienced test engineer, knowing that a fault is trapped in R_3, would be to set the select bits $S_{3\text{-}0} = (1,1,1,1)$ and clock the circuit until the value in R_3, including the fault effect, propagates through all the registers to the output $DO_{7\text{-}0}$. By using this lookahead strategy, the targeted fault is propagated forward, but in the process other faults may also be flushed out of the circuit. However, if a trapped fault reaches R_{16}, it is at a dead end unless R_S has already been set to select R_{17}.

Dead ends are a major problem for sequential test pattern generation. In the example just given, the 16-stage shift register, the trapped fault remained alive, it just didn't go where it was expected to go. More often, the trapped fault gets blocked in the combinational logic between the flip-flop in which it is trapped and the destination flip-flop. In the fault simulator, the fault is converged at the point where it becomes blocked. In general, whenever a fault is being sensitized or propagated, an effort must be made to sensitize and propagate simultaneously.

Consider the circuit in Figure 12.13. A fault is sensitized at the input of a NOR gate. When a clock edge is applied, the $\overline{D}$ will be clocked into a flip-flop where it will become trapped. However, another flip-flop receives a 1 when the clock is applied. Unfortunately, that 1 becomes inverted and blocks the $\overline{D}$ from propagating any further. In order to successfully propagate the fault effect in this circuit, it must simultaneously be *sensitized* in the combinational logic and *propagated* through the logic corresponding to the next time image. When that happens, requirements will be imposed on the destination flip-flops (the bank of flip-flops on the right). These requirements will then have to be justified simultaneous with the sensitization of the stuck-at fault. Note that a 1 was assigned to one of the flip-flops in the left bank in order to justify a 0 from the NOR gate. That assignment can be changed to a 0, and the other input to the NOR gate can be assigned a 1.

An alternate solution, if the flip-flop that receives a $\overline{D}$ has a hold mode, is to force that flip-flop into the hold state. But, that also requires looking ahead. In this case, rather than look ahead into the next time frame, the state search must simultaneously

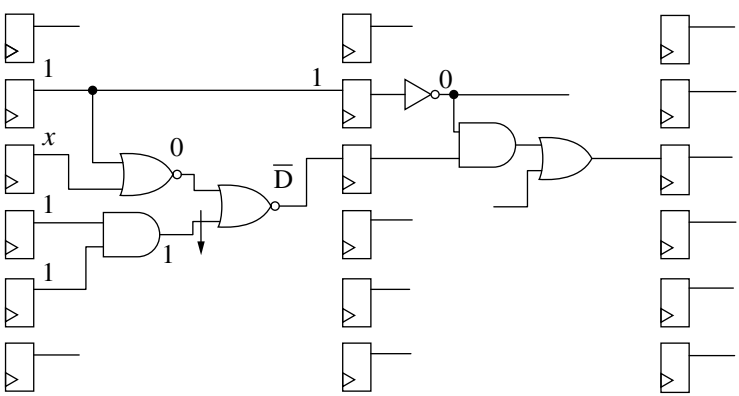


Figure 12.13 Encountering a dead end.

sensitize the fault and justify the hold mode for the target flip-flop. If the trapped fault can be held in the target flip-flop for an indeterminate number of time frames, then a propagation path can be set up while the target flip-flop retains the fault effect. Eventually, the trapped fault propagates forward. Of course, if the destination is not a primary output, the same considerations hold at the new destination; that is, the fault could be at a dead end if care is not taken to hold it until a propagation path is established.

When trapped faults are selected by the fault simulator and passed on to the Supervisor, the corresponding RTL level storage elements in which they are trapped are identified by means of a map file. The Supervisor then selects from among the heuristics. The fault chosen for propagation may be trapped in a data path, or it may be trapped in control logic. If it is trapped in a data path, then the object is to propagate it forward toward an output. If the fault is trapped in control logic, then it can usually only be observed indirectly by means of its effects on the data path elements. For example, suppose the fault-free circuit is attempting to perform a logic AND on two arguments *X* and *Y*, and a fault in the control section causes an OR operation to be selected. Then, for the values $X_i = 0$ and $Y_i = 1$, the fault-free circuit responds with a 0 and the faulty circuit responds with a 1.

The propagation in this case is not done by chance. BATG must be able to recognize whether a fault effect currently being processed is in control logic or data-flow logic. Control logic includes such things as status registers and mode control registers. For example, suppose a particular mode control register determines the display resolution and number of colors chosen by a graphics chip. Such a register is often write-only; it cannot be directly read out. In order to determine its contents, the data coming out of the graphics chip must be inspected. If a defect exists in the mode control register, data will come out at the wrong rate, or the wrong data will come out, in which case the defect will be identified. BATG must have enough intelligence built into it to enable it to understand, at some level, that it must set up data registers with values that can cause incorrect values in a mode control register to appear at the output pins of the chip in the form of incorrect data, in order for control register faults to be identified.

Another example of indirect identification of register bits occurs when a status register for an ALU must be checked (see Figure 3.1). To determine if an overflow occurred during an ALU operation, a conditional jump instruction is executed. If an overflow is supposed to occur during an ALU operation, then the jump address should appear at the address bus. If the next sequential address appears, the overflow bit of the status register must be stuck-at the wrong value. In this case, BATG must set up the processor to first perform an ALU operation, and BATG must determine what arguments are needed to either induce or avoid an overflow, depending on which of these conditions is being checked.

Trapped fault selection is one of the activities that can be guided by heuristics. In the early phase of test pattern generation, it is usually desirable to flush out as many faults as possible, as quickly as possible. This can help to reduce fault simulation time, and it can help to avoid performing complex searches on faults that would normally fall out as a byproduct of other searches. Selection criteria include ease or

difficulty of flushing out trapped faults. If a large register, say a 32- or 64-bit data register, has many trapped faults, then it is a candidate for propagation. If two or more registers have comparable numbers of trapped faults, then another level of decision must be employed to further refine the decision process.

One of the difficult things to do in an ATPG program is to make judgment calls. In the shift register of Figure 12.12, a comprehensive test strategy needs to consider how many faults can be detected by propagating a value through the entire shift register. If fault coverage for the register is high and only a few faults are undetected, there may be no benefit in adding many clock cycles to the test in order to propagate a value through the entire shift register. Another complicating factor is the level of effort required to set up the values required at $S_{3\text{-}0}$. It was assumed that it could be done in one clock cycle. In reality, $S_{3\text{-}0}$ may require that a state machine traverse many states in order to reach the state that enables the needed values onto $S_{3\text{-}0}$. This is an area where TDX could have benefited from a rule-based system, invoking the system to make decisions based on current fault coverage percentage, ease or difficulty of sensitizing and propagating a fault through the RTL, payback in estimated number of additional fault detections, and so on.

To determine how to rate trapped faults in terms of difficulty, it must be possible to link error signals back to their fault origins. The fault origins, in turn, are linked to the input or output of the gate at which they originate. From there the controllability and observability numbers at the gate input or output can be used to estimate a level of difficulty, hence a value, for that fault. It should be noted, however, that the mechanical computation of C/O numbers does not always provide an accurate indication of the ease or difficulty in controlling or observing the fault. It is possible for faults in control sections and the data path to have similar C/O numbers, but the effects of faults in the data path are directly observable, while faults in control logic are indirectly observable; that is, they are detected by observing their effects on operations performed in the data path.

If it can be determined that many faults in a control section are trapped in the register, then the register contents can be given a high value during the selection process. To determine whether a fault effect originated in the control or data path part of the circuit, the data structures can be examined by tracing from the gate associated with the fault origin back to the flip-flops that drive that gate and forward to the flip-flops that are driven by it. From the map file these are easily associated with their RTL level counterparts, which can then be related to their purpose in the circuit.

As we saw in preceding paragraphs, trapped faults, a seemingly innocuous concept, can introduce many complexities into the equation when all the issues are considered. In real-life circuits, many flip-flops and registers are buried deep in the circuit and require many clock cycles to control and observe. Others may be easy to control and difficult to observe or vice versa. When considering trapped faults, it would be useful to know in advance which of the flip-flops and registers are easy to control and/or observe. A register may have many trapped faults that are desirable to propagate to the output, but it may be the case that it is exceedingly difficult to propagate the contents of that register to the output.

Conversely, the contents of a register may be easy to propagate to the output. It may, in fact, directly drive an internal bus that is connected to an output port. Part of the task of TDX was to learn about the circuit. The controllability and observability (C/O) numbers generated by the testability analysis program were a first-stage attempt to understand C/O issues. From there, other means were used to evaluate the ease or difficulty of propagating values to outputs. In effect, BATG was constantly refining its understanding of how the circuit behaved, and how it could be controlled and observed.

As fault coverage increased, heuristics for selecting trapped faults often changed. If BATG learned how to sensitize and propagate faults in a particular area of a design, it might be desirable to continue developing test sequences for that area until all or nearly all the faults in that region become detected. An alternative may be to address faults in a function for which there is little or no coverage. The rationale for this is to get overall fault coverage up to some desirable level with the least number of vectors. This is motivated by the fact that the user may want to hold down the overall test length (cost) while getting the best possible fault coverage within that test length constraint. Also, as has been pointed out in the literature, test quality is influenced to some extent by how well fault coverage is distributed.[24] When fault coverage reaches some predefined level, it is possible at that point to begin attacking individual faults, or small clusters of faults, with more refined heuristics.

12.8.2 DEPOT

By virtue of its architecture, TDX could propagate and justify values derived from the RTL model by means of error modeling, or it could propagate and justify stuck-at faults identified by a gate-level ATPG that could determine what state the circuit had to be in for a fault to become sensitized. To that end, a gate-level combinational test pattern generator was developed. It supported the D-algorithm and a variant of PODEM. The gate-level ATPG was called DEPOT (DEductive, Path-Oriented Trace).

Conceptually, when DEPOT was running, TDX, to all appearances, behaved like any other scan-based ATPG, at least for synchronous circuits. It selected an undetected fault, then worked its way forward to a flip-flop or primary output, and justified assignments back to primary inputs and/or flip-flops. However, at this point the similarity to a scan-based system ceased. A priority for DEPOT was to sensitize a fault with the smallest possible number of state assignments. Sequential state searches were costly in terms of computations, and the greater the number of state assignments, the greater the search space, and the greater the likelihood of conflicts.

Because PODEM was given a list of inputs (primary inputs and flip-flops) to which assignments were to be made, and these inputs were selected by tracing back from assignments that required justification, it would often make assignments that were not essential to sensitizing a fault. It turned out that, for the purposes of generating the smallest list of assignments to flip-flops and I/O pins, the D-algorithm generally proved to be more frugal. SCOAP numbers were used to control justification and sensitization, and these numbers were more effectively used by the D-algorithm.

Another priority for DEPOT was to try to match the existing state of a circuit. If two or more sensitization solutions exist and if one of them more closely matches the current state than any of the other solutions, then it is usually the more desirable solution, since fewer goals are generated. It is possible, however, that only one storage element needs to be changed from its existing state in order to sensitize a fault, but it may be extremely difficult to control. A cost function involving heuristics, including controllability/observability numbers, helped to make a decision in those cases.

An optimal strategy was to look for easy solutions. For example, a sensitized path may already exist for an undetected fault from its origin to the data input of a flip-flop. Since the concurrent fault simulator had a complete record of fault effects, it could examine logic gates driving flip-flop inputs, looking for fault effects that corresponded to undetected faults (cf. Figure 3.10). If one or more such fault effects were found, then toggling the clock would cause that fault, and possibly others, to become trapped.

Strategies that were under consideration (but not implemented) included Boolean differences and binary decision diagrams (BDD). Given a fault to be sensitized in a particular cone, the object was to find a closed form expression sensitizing that fault within the cone (cf. Section 4.13.1). The expression could then be evaluated analytically, relative to the current state of the circuit, to find the best sensitization state. The best sensitization state might be one that most closely matches the current state of the circuit, or it might be one that is deemed least expensive (easiest), based on some cost function. If a fault exists in two or more cones and if closed-form expressions could be generated for each of the cones, a more comprehensive cost function could be implemented.

Section 12.6.3 introduced the concept of primitive function test patterns (PFTP) for members of the library of parameterized models (LPM). It was pointed out that a comprehensive set of vectors, based on the functionality of individual members of the LPM, has the advantage that all inputs to an n-wide data port can be assigned simultaneously, permitting more faults to be detected, or more classes of faults to be addressed, such as shorts between adjacent pins to the function. These vectors can be used in place of vectors that were generated by DEPOT for specific faults, or vectors generated by DEPOT could be merged with these vectors from the library. Another option is to use the PFTP vectors first and then, if faults escape detection, use DEPOT to target those faults that remain undetected.

History files were another TDX feature. Information useful in a history file included a record of successes and failures while trying to drive a circuit into a specific state. It was found that success or failure in reaching a target state often depended on the current state of the circuit. Sometimes the target state could be reached merely by toggling the clock. At other times long, complex sequences were required. It proved useful sometimes to tag a particular difficult-to-reach state to indicate that if it were reached while trying to achieve some other goal, it should then be considered for exploitation. This is one of those examples of trying to develop rules that mimic behavior of the human engineer who, while developing sequences to either verify a design or create manufacturing test programs, may

break off a particular approach and pursue another target of opportunity that appears to give a greater payback with less effort.

The history file is also useful when analyzing closed-form expressions, such as those obtained from boolean differences, for identifying preferred sensitization states. History files can become enormous, so they must be limited to key control constructs such as state machines, mode control registers, and status registers. History, together with controllability and observability values for these registers, can then become part of a more global evaluation process. This higher level of analysis provides a payback when what looks like a less expensive solution turns out to be the more expensive solution, or vice versa.

12.8.3 The Fault Simulator

Since the original intent of TDX was to generate stimuli for manufacturing tests, a fault simulator was needed to compute fault coverage and to identify undetected faults. It was a full-timing, concurrent fault simulator, able to accurately fault simulate both synchronous and asynchronous circuits. The simulation engine supported both an event-driven engine and a read/write array for processing zero delay elements. If the elements of a combinational block of logic all had zero delay, they would be rank-ordered. This provided some of the benefits of cycle simulation, with a payback magnified by the fact that rank-ordering not only reduced the number of logic event evaluations, but also reduced the number of fault event evaluations, and there tended to be, on average, about 10 times as many of these evaluations.

The fault simulator was able to fault-simulate subsequences provided by the Supervisor, then return control to the Supervisor. After fault simulation the TDX Supervisor would then request that the fault simulator identify a trapped fault for propagation, in which case BATG would be invoked to perform RTL level propagation of the fault; if there were no trapped fault candidates, the Supervisor would select a fault from the list of undetected faults and invoke DEPOT.

If there were several trapped faults, the Supervisor could identify particular registers or flip-flops in which it was interested in trapped faults, or it could request that the fault simulator return a linked list identifying all of the storage devices that contained trapped faults. Identifying trapped faults in particular registers or flip-flops was often more valuable during the early stages of the run when it was likely that all or almost all of the storage devices would have trapped faults. During this stage, general-purpose registers might hold many trapped faults as a result of ALU operations. As the run progressed and fault coverage increased, the distribution of fault effects became more sparse, and the likelihood of finding trapped faults would decrease in inverse proportion to the fault coverage.

The fault simulator was also used as a logic simulator. In this mode it ignored the fault effects. After a sequence of vectors was generated, the simulator logic simulated them to determine if the correct destination state was reached at the end of the sequence. If the sequence caused the circuit to behave as intended, then the sequence would be fault-simulated to (a) identify faults that were detected by the subsequence and (b) identify trapped faults. If the sequence failed to drive the

circuit into the desired end state, then the sequence could be abandoned, or an attempt could be made to repair the sequence, (see Section 12.8.12, Learn Mode). During this operation it was necessary for the simulator to avoid processing fault effects. It was also necessary for the simulator to save the circuit state prior to evaluating one of these subsequences so that it could restore the circuit state in order to logic simulate another subsequence if one needed to be evaluated, or to fault simulate a subsequence.

12.8.4 Building Goal Trees

We briefly review the concept of goal trees and searches. Figure 12.14 describes a circuit in terms of (some of) its storage elements, which contain values representing current state. A gate-level ATPG such as DEPOT is employed to find a sensitizing state for a selected fault. The sensitizing state, represented as the goal state in Figure 12.14, usually differs from the current state. Behavioral search routines explore the RTL in order to find a sequence of input vectors that cause the circuit to transition from the current state, or from a reset state, to the goal state.

In the SCIRTSS system, a gate-level ATPG was used to find several, or perhaps all, possible sensitization states for a given fault. Then the search routines tried to justify as many as possible of these states. That could be seen in the Petri net example given in Section 12.7.2. The place labeled goal had arcs from two transitions. Each of the transitions corresponded to a sensitization state determined by the gate-level ATPG.

In SCIRTSS, the creation of goal trees using multiple sensitization states was acceptable. However, in TDX it was found to be impractical. As circuits grew in size and complexity, the number of sensitization states became prohibitively large and goals became quite complex. An individual sensitization goal state might require justifying several of the storage elements shown in Figure 12.14. In addition, the number of time frames required to create a bridging sequence from the current state to the goal state might exceed the maximum permitted. Additionally, many of these sensitization states provided by the gate-level ATPG were quite similar, differing perhaps in values selected from some data path element such as an ALU. In this case, the goal trees were essentially the same, and the goal building process was repetitious if all such sensitization states were pursued in parallel.

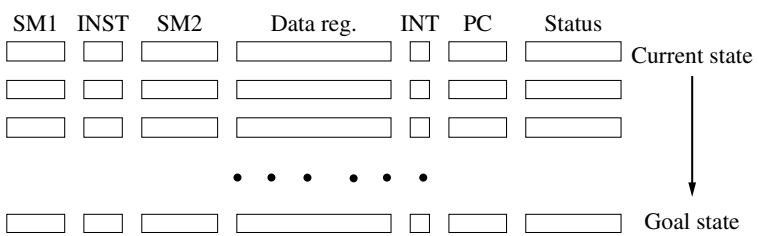


Figure 12.14 Creating a bridge from current state to goal state.

DEPOT would therefore be programmed to search for the most economical set of goals based on C/O numbers. If that state could not be justified by the search routines, then DEPOT would be invoked with a request to find an alternate sensitizing state. Earlier it was mentioned that the C/O numbers were based on a SCOAP-like program. The sensitizing state usually required assignments for two or more resources, such as, for example, a state machine and a data register and a status register. It might be the case that one of them was easy to control and/or observe, while another might be very difficult to control and/or observe.

Furthermore, there was an implicit assumption that C/O numbers were based on starting from a reset state, while, in reality, the state search routines attempted to build a bridge back to the current state. In many cases the best C/O numbers did not facilitate creation of the best bridge from goal state to current state. As a result, it was found that the C/O numbers, statistically, would give overall improvements in performance, but there was no certainty that they would provide the best solution in individual instances.

12.8.5 Sequential Conflicts in Goal Trees

Earlier in this section we discussed problems caused by faults that could not be propagated; these were called dead ends. Here we examine another fundamental problem with sequential ATPG. In this case the problem is one of justification. Referring back to Figure 12.14, suppose that a particular fault becomes sensitized as a result of SM1 being in state 0010 and SM2 being in state 010. Suppose also that the instruction register INST must contain the value 01011101 and that the status register must contain XXXX01XX; that is, Status[3:2] must contain the value 01. Suppose that three of the four requirements are satisfied and that the only requirement not satisfied is the requirement that INST = 01011101.

In a combinational circuit the requirement that a PDCF for a four-input AND gate contain the values (1,1, 0,1) might be considered an analogous condition. However, in a combinational circuit, if there is a solution, then all of the values will be justified in the same time frame. While propagating forward to an output, additional requirements are added in order to extend the sensitized path. These assignments, too, will be satisfied in the same time frame. For a sequential circuit the situation is very different. Current state of individual registers may match goal requirements, or some, possibly all, of the goals may differ from current circuit state. Satisfying these goals is seldom as simple as backtracing through combinational logic.

In the example from Figure 12.14 just described, if one considers only the fact that three of the goals are satisfied, it would be tempting to ignore those three goals and only be concerned with the unsatisfied goal. However, the human, recognizing that we are dealing with a CPU and that the instruction register and the state machines are inextricably bound up and interdependent, will deal with the issue in a more holistic way. The human will first try to satisfy the status register, totally ignoring the goals that are currently satisfied. It may well be the case that Status[3:2] represents an overflow condition. If Status[3:2] is not satisfied and if the INST value represents a jump on overflow (JOV), then the overflow condition must exist in order

for a jump to occur. So it makes no sense to attempt to justify the JOV instruction if the overflow status is not set.

Other circuits may be more benign. Given n goals that must be satisfied, the goals may be completely independent, meaning that their states do not depend on one another, and it may be possible to satisfy some or all of them simultaneously. In that case, the obvious choice, when given a set of goals to satisfy, is to select only those registers and flip-flops that are not currently satisfied and build goal trees targeted at those goals.

There is yet another possibility: It may be the case that several goals must be satisfied, and they are not mutually dependent, but rather depend on common resources. This is depicted in Figure 12.15. In this example there is an AND gate driven by bits from each of three registers labeled R_1, R_2, and R_8. DEPOT is given the assignment of finding a circuit state to sensitize a stuck-at-1 fault on the middle input to the AND gate. DEPOT comes up with the assignments (1, 0, 1) on the three registers. At this point the TDX Supervisor passes control to BATG, whose job is to figure out how to build a sequence that drives the circuit from the current state to the goal state so that R_1, R_2, R_8 = (1, 0, 1). Note that the configuration described here is fairly typical in devices such as peripheral controllers and video controllers.

When BATG starts building goal trees, it backtraces from the individual registers. Assume, without loss of generality, that R_1 is chosen to be processed first, followed by R_2 and then R_8. Consider how a programmer might cause the required values to appear in the registers. In order for R_1 to be justified, the programmer must first cause the signal WE to become enabled. Then INDEX REG, an 8-bit register, must be loaded with the correct value from the data bus in order to select R_1. The programmer may then read out the current value in register R_1, alter a single bit using a mask,

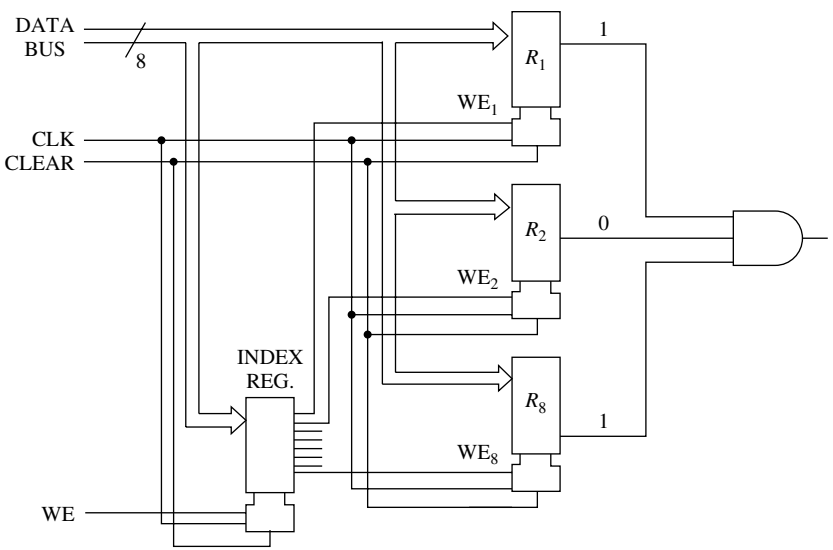


Figure 12.15 Common resources.

then reload R_1 with the updated value. This complete operation is then repeated for R_2 and R_8. In fact, the programmer might simply write a function or macro to alter the bit(s) of interest in the registers. The key point to note is that these operations take place serially, because the same resource, namely, INDEX REG, is required for each of these operations and its value must be different for each operation.

In building goal trees, BATG starts from the destination. If there is only one goal, say R_1, BATG would not have much trouble satisfying the goal. While the programmer must read out the value in the register in order to mask it and alter only the bit(s) of interest, BATG can inspect the circuit model to determine the current contents of R_1 and then stuff an altered version of that value onto the data bus. Now, when BATG selects the next goal from which to backtrace, because a different value is required in INDEX REG, a conflict is going to occur. Recall, from discussion of the D-algorithm, that when a conflict occurs, a decision is voided and an alternate choice must be pursued. In the example being considered here, a conflict appears to occur because BATG is trying to load three registers—R_1, R_2, and R_3—with different values from the same data bus at the same time.

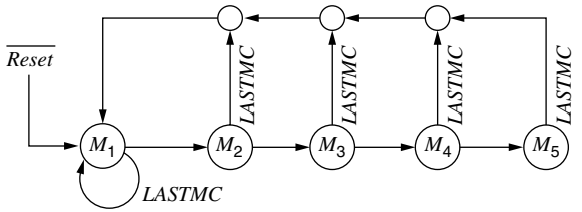
A key point in solving this set of goals is the recognition of two factors: First, the three goals R1, R2, and R8 are not *interdependent goals*; that is, satisfying one of them does not depend on, or require, any particular value in either of the other two. Second, the *subordinate goal*, INDEX REG, is independent of all three goals. The value of this observation lies in the fact that, since the goals do not depend on one another, they can be processed serially. The goal subtree for one of the registers will not disturb the goal subtree for the others, as long as the goal subtrees do not attempt to use the same resources in the same time frame.

12.8.6 Goal Processing for a Microprocessor

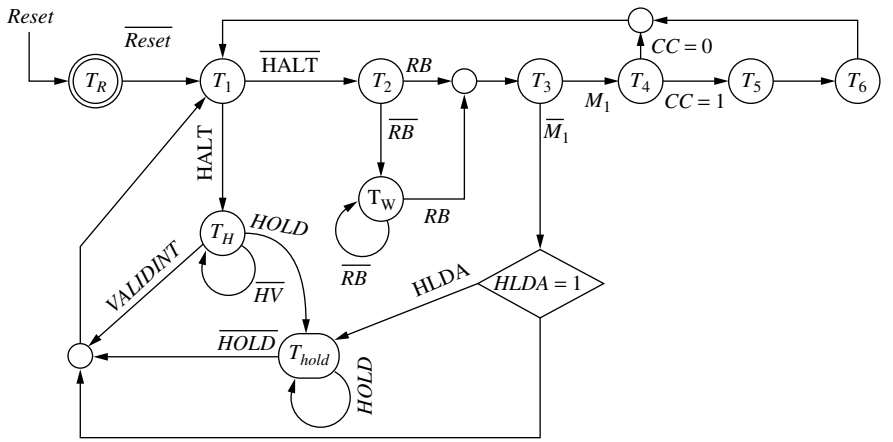
We consider again the goals in Figure 12.14. The state machines in Figure 12.16, from an 8-bit microprocessor, will be used to illustrate how these goals are interrelated. The major state machine, Figure 12.16(a), has states M_1 through M_5. An instruction may require anywhere from one to five *M*-cycles, each of which is broken down into three or more *T*-cycles. Instruction fetch (*I*-fetch) is accomplished during T_1 through T_3 of M_1. If memory data are not ready, the state machine transitions to T_W and waits there until memory provides a data ready signal.

The first *M*-cycle traverses at least four, and possibly six, *T*-states, depending on the op-code. Other *T*-states include T_{HOLD} and T_{HALT}, although, in the interests of brevity, they will not be considered further. The remaining *M*-cycles always contain three *T*-states that are mainly used to move data to and from memory.

The JOV instruction discussed earlier is made up of three machine cycles. The op-code is contained in the first byte, followed by two additional bytes that contain the jump address in the event that a jump is taken. The op-code is fetched and decoded during the first machine cycle. The next two machine cycles are used to fetch the jump address. If the jump condition is met, then the address contained in bytes two and three of the instruction are put out on the address bus. If the condition is not met, then the PC (program counter) is put out on the address bus.



(a) Major state machine



(b) Minor state machine

Legend:

CC - Number of clock cycles in first machine cycle
*M*1 - Machine cycle 1
RB - READY + BIMC
BIMC - Bus Idle Machine Cycle
HV - HOLD + VALIDINT

Figure 12.16 Interacting state machines.

Suppose that a test is to be created for a fault wherein the fault-free circuit executes a jump if the overflow bit is set, and the faulty circuit proceeds to the next instruction in sequence. Then a set of goals includes a requirement that the overflow bit of the condition code register be set, the JOV instruction must be present in the IR (instruction register), and, by virtue of sensitization requirements imposed by DEPOT, the microprocessor will have to be driven to a specific *M*-cycle and *T*-state. Also note that, following the sensitization phase, there must be a propagation phase during which the actual jump occurs; that is, the address in bytes two and three of the instruction are placed on the address bus. However, in this discussion we will confine our attention to the sensitization phase.

Now the question is, What happens when an attempt is made to satisfy the sensitization goals? Since several goals must be satisfied, the first question is, Should the backtrace proceed goal by goal, or should all of the goals in a given time frame be processed in parallel before backing up to the previous time frame? If goals are processed one at a time, backtracing until it is satisfied, then processing the next goal, conflicts may not be recognized until long after most of the goals appear to be satisfied. In a given situation, nine out of ten goals might be satisfied, only to find that the tenth goal is irrevocably in conflict with one or more of the others. Alternatively, if the goals are processed in parallel, "apparent" conflicts at the start of processing may cause the search to be abandoned, even when there is a solution if the goals are processed serially.

Consider what happens when the goals are processed serially. Suppose the first goal selected requires that the minor state machine be in state T_3. This turns out to be quite easy. In fact, there is a good chance that the circuit is already in state T_3. The next goal chosen may require that the major state machine must be in M_3. This presents a more significant challenge, since the signal $M1$ in Figure 12.16 must be set to 1. The RTL description must be examined in order to identify those instructions that have three (or more) machine cycles. Then, one of those instructions must be selected.

The ideal situation would be to have the JOV op-code selected, but that requires the search routines to recognize, while searching for an instruction with three or more M-cycles, that the JOV is the only candidate not in conflict with other goals. Humans do this fairly easily and fairly regularly, since humans recognize that they are dealing with a microprocessor, and it fits a particular behavioral paradigm. Humans also instinctively recognize that what works for a microprocessor may be a totally wrong approach for some other category of circuit, such as a digital signal processor or a video controller. It is important to note that the human is often relying on many years of formal training, as well as considerable experience, to help him or her make decisions about which goals should be satisfied first. In fact, the human is often not even aware that he or she is thinking in terms of goals. Goal ordering and prioritizing is done instinctively and subconsciously.

As part of this training and experience, the human imposes different levels of understanding on different types of circuits, recognizing from experience and intuition the very different nature of a peripheral chip, in contrast to a microprocessor. This may motivate the human to start by devising an overall strategy. A software program must formalize this knowledge in order to emulate the human. And, just as the human makes informed guesses about how a particular circuit might behave under different circumstances, the computer program must employ heuristics that serve as its counterpart to the human's educated guesses. This requires a considerable amount of preprocessing on the part of the program to achieve a level of capability comparable to a human.

For the problem presented here, a human recognizes that the first order of business is to set the overflow bit in the condition code register. The second step is to load the instruction register with the JOV op-code. Then, finally, maneuver the state

machines into the required states. One way that a program might recognize this priority of events is to keep track of the number of times that each storage element switches while simulating the subsequences. Those that switch often can be judged to be easy to control.

Those that have never switched can, at least temporarily, be judged to be difficult to control. This is essentially a heuristic—that is, a criteria that may not be completely accurate—but may nevertheless frequently prove to be useful. For example, the condition code bits may rarely switch, while the state machines constantly change state. This may suggest that the condition code register should be the first goal to be processed.

To help determine the ease or difficulty of controlling variables, TDX had an initialization mode. In this mode, TDX first attempted to initialize every storage element or group of elements in the circuit. Devices were grouped whenever there was a logical relationship, as when they formed a state machine or an *n*-wide register. In a clean circuit—that is, a synchronous circuit where all, or nearly all, storage devices had a set or clear line—it was relatively easy to initialize those storage elements. Devices that could not be directly set or cleared then become individual goals. After all of the storage elements had been processed, an attempt was then made to switch every storage element in the circuit. Again, every logical group in the circuit (e.g., state machine or *n*-wide register) was treated as a single goal.

If TDX could not initialize a storage element, or cause it to switch from its current state, the element would be flagged as uncontrollable. Since elements that could not be controlled were often critical to the controllability of other storage elements, they would be identified to the user. Sometimes an element could not be controlled because it depended on another element that was, in turn, uncontrollable. By identifying a root cause—that is, a single element that directly or indirectly controlled other elements—it was possible to minimize the number of uncontrollable elements. By forcing the root cause element to a 1 and 0, it was possible to determine conclusively whether the element depending on the root cause was, itself, controllable or uncontrollable.

The user could place the uncontrollable elements into a list that could then be used as the basis of a partial scan chain. Another option was to maintain a list of the uncontrollable elements and periodically check them whenever simulating sequences. If a storage element should happen to enter a known state, an attempt was then made to decipher the applied sequence and try to determine what sequence of events caused the storage element to either transition from an X state to a known state, or switch from a known state to the opposite state.

Another option was to ask the user to provide a sequence that could cause the targeted storage element to achieve a desired state. This sequence could then be stored as part of a history file and retrieved as needed. However, it should be noted that such a sequence, if it starts from the reset state, may not achieve its intended result when entered from some other circuit state. In fact, when used with other goals, such a sequence could be counterproductive since all the work done to generate sequences that satisfy other goals may be completely wiped out if the reset line is exercised. In general, depending on the reset line to help satisfy goals can be a bad practice.

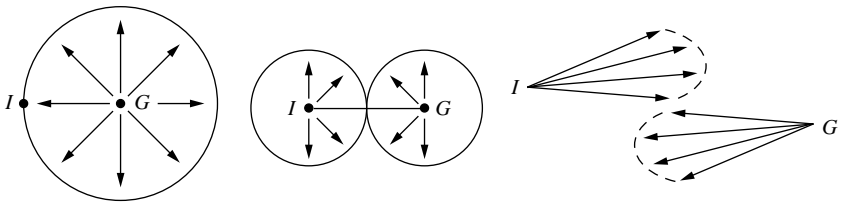
12.8.7 Bidirectional Goal Search

Working backwards from a set of destination goals can cause many conflicts. Goals competing for the same resources (e.g., INDEX REG in Figure 12.15) cause what appear to be unresolvable conflicts. The problem is that goal searches, like ATPG algorithms, do not handle the time domain very well. It is sometimes easier to maintain consistency among goals when starting from the current state of the simulation model and building sequences forward in time. In Figure 12.15, when attempting to justify R_1, R_2, or R_8, it is quickly seen that one of these registers must be selected by INDEX REG, so INDEX REG becomes a subgoal. From the RTL it is recognized that, for INDEX REG to be loaded, WE must be set to its enabling value. Once INDEX REG is loaded, the selected register can be loaded on the next clock. In some circuits it may be easier to see potential conflicts when working in a forward direction. A natural inclination for an ATPG when justifying goals backward in time is to try to force a resolution of the conflicts between R_1, R_2, and R_8.

It is still important to recognize that R_1, R_2, and R_8 are independent goals, and INDEX REG is subordinate to each of the registers. A forward search that does not recognize this and tries to satisfy all three concurrently will fail. This is a circuit configuration where it is possible to recognize that the registers are in conflict. The select lines choosing one of those registers will likely appear in a common construct, probably a case statement, in the RTL. From there it is possible to tag them in the knowledge base with data identifying the fact that they are mutually exclusive. Then, while the goal tree is being constructed, it can be searched to determine if any conflicting goals exist in the tree and if they are actually in conflict. This last condition needs to be considered because it is possible that two conflicting goals exist in a goal tree, but they are sufficiently displaced in time that they do not conflict.

Up to this point, we considered building a sequence by starting from the target goal set and by starting from the initial, or current, circuit state. A third option is to build a bridging sequence to transition from current machine state to the desired goal state by working from both directions simultaneously. When doing a bidirectional search, generally fewer nodes have to be expanded.[25] This is suggested in Figure 12.17. A breadth-first search is performed from the goal state G, extending out in all directions until it reaches initial state I. In the bidirectional search, node expansion takes place from initial state I and goal state G simultaneously. In a typical tree, the number of nodes at each level of the tree increases from the previous level since each node at a given level is being expanded as the tree descends. Hence a tree of n levels, growing broader at leach level, will have more nodes than two trees, each of level $n/2$.

When employing heuristics to narrow the search, it would be expected that this phenomenon would continue to hold. However, the heuristics may be too effective. Figure 12.17(c) illustrates a situation where a bidirectional search is performed with the aid of heuristics. The heuristics help to narrow the search, only pursuing choices that appear to be optimal. This is done when searching both from the initial state I and from the goal state G. Because the search is narrowed, the two search cones in this example bypass each other. One possible outcome is that the attempt to build a



(a) Breadth-first (b) Bidirectional (c) Heuristics-guided bidirectional

Figure 12.17 Bidirectional search.

bridging sequence from the initial state to the goal state may be aborted before a solution is achieved. Another possible outcome is that one or the other of the two cones may eventually terminate at the desired destination, but the cost of the bidirectional search may be greater than if a unidirectional search had been performed.

12.8.8 Constraint Propagation

When building goal trees, it often happens that a goal does not explicitly dictate a value for some circuit element. Rather, the goal requires, for example, that the value in register *A* be greater than the value in register *B* in order to establish an overflow bit in a status register. This is referred to as a constraint. The constraint may already be satisfied, in which case it is necessary to maintain the relationship. This is referred to as *constraint propagation*. This can become problematic because, while exploring the logic to satisfy other goals, the relationship may become negated.

Alternatively, the constraint may not currently exist and may need to be satisfied. The constraint may be inextricably bound with other goals in such a way that it has to be satisfied at the right point in a sequence; in some cases it may need to be satisfied first, in other instances it may be necessary to satisfy it after other goals have been satisfied. Trying to force it to take effect as, say, the last goal to be satisfied from a group of goals may disrupt other goals that have already been established.

Trying to determine how to satisfy a constraint can be a difficult problem. Consider again the example of register *A* and register *B*; suppose it is necessary to perform a subtraction in order to produce a negative result. Which register should be the minuend, and which register should be the subtrahend? If registers *A* and *B* are both general-purpose registers, such that either could be the minuend and subsequently contain the difference, the decision could be quite different from the decision if register A is an accumulator, requiring the difference to end up in *A*. As a further consideration, if *A* is an accumulator, it must be confirmed that the results will still be there at the end of the goal-building process.

It may be the case that the outcome of the operation sets a flag bit in a program status word (PSW). Then, it becomes necessary to protect the PSW whenever other goals are being processed to ensure that the PSW is not inadvertently altered while other parts of the goal tree are being constructed. Sometimes a PSW gets modified

as a result of a POP instruction upon returning from a subroutine. This is another of those situations where the human recognizes unintended side effects, but the computer program may not have been developed to the point where it checks for all these exigencies.

12.8.9 Pitfalls When Building Goal Trees

Many seemingly innocuous choices that are made when exploring RTL code can lead to major problems. Toggling a reset line at the wrong time can destabilize an entire sequence of goal trees built up over many time frames. In a data base there may be a data structure that describes how to control the inputs to a register in order to load it with a particular value. One entry may assert that the register can be set to the all-0 state by setting a signal called *CLR* to 0. Unless some preliminary investigation has been performed, identifying *CLR* as a reset state, the ASCII string "CLR" has no more significance than any other signal string. It is only when *CLR* is asserted that the consequences of toggling it to 0 are realized. The implications of toggling a *CLR* line can often be realized from the RTL code. As an example, consider the following Verilog code:

```
always @(posedge CLK or negedge CLR)
begin
  if (CLR == 0)
    RegA = 0;
  else
    RegA = DBUS;
end
```

This behavioral Verilog code is commonly used to reset a flip-flop or register, or to load it from some source. In this case the source is a bus called *DBUS*. It is likely that there will be many more storage elements instantiated in this manner. The *CLR* signal will cause all of them to be reset simultaneously. The general nature of this behavioral Verilog construct, along with the fact that the *CLR* signal is in the sensitivity list, is a tip-off that it has some significance beyond its ability to load a 0 into a register. The additional fact that *CLR* probably has a large number of fanout points is a further clue that it should be approached warily when building goal trees.

When building goal trees from RTL code, many choices exist. Consider the following Verilog code:

```
always @(Sel or D0 or D1 or D2 or D3)
begin
  case(Sel)
    00: Mxout = D0;
    01: Mxout = D1;
    02: Mxout = D2;
```

```
    03: Mxout = D3;
  endcase
end
```

In this 4-to-1 multiplexer, *Sel* determines which source is connected to *Mxout*. This structure will be represented in the knowledge base, together with information that reflects the ease or difficulty of controlling the four input sources, D_0, D_1, D_2, and D_3. One of the problems with this controllability information is the fact that it is based on static analysis of the circuit. As an example, to get a desired value at *Mxout*, controllability analysis may assert that D_1 is the best choice. But, in the current state of the circuit, D_3 may be much easier to justify. Another problem with controllability information is the fact that the same paths tend to get exercised repeatedly.

By randomly choosing sources to connect to *Mxout*, there is the possibility that other previously unexercised logic will get exercised; as a result, a more thorough exercise of the entire circuit will result. Note that this tendency to repeatedly exercise the same data paths is also a human tendency. This should not be surprising because when a human is focused on setting up a particular sequence of events, he or she does not want to be distracted by having attention drawn away to another subproblem involving complicated choices that cause the immediate goal to be obscured. If choices are made based on heuristic information, there is the possibility that the heuristics are self-reinforcing because of successes, rather than because they represent the best choices. This behavior, too, has its counterpart in human behavior. We tend to continue to do things that worked for us last time, rather than explore new paths that might yield an even bigger payoff.

12.8.10 MaxGoal Versus MinGoal

It was mentioned earlier that SCIRTSS would build a goal tree in which the top level could be the OR of several goals. This is illustrated in Figure 12.18. Three top-level goals G_1, G_2, and G_3 represent three different sensitization states, based on three different results while backtracing in order to justify a PDCF. Each of these top-level sensitization goals G_i could require several subgoals S_i to justify it. SCIRTSS would attempt to justify all of the top-level goals. However, when TDX was being developed, using circuits for which the RTL represented many thousands of gate equivalents, it was too costly to attempt to justify all of the top-level goals. Hence, controllability and observability information was used to try to find the single most economical top-level goal.

Given a top-level goal G_i, the subgoals S_i could themselves represent a very costly solution in terms of the number of vectors needed to build a bridge from current circuit state. Hence, during the development of TDX, one of the experiments performed was to compare the effectiveness of a MinGoal strategy versus a MaxGoal strategy. TDX examined the goals S_i that were needed to sensitize a fault or propagate a trapped fault, and it determined which of these goals were already satisfied. Consider the circuit in Figure 12.15, where R_1, R_2, and R_8 had to be justified.

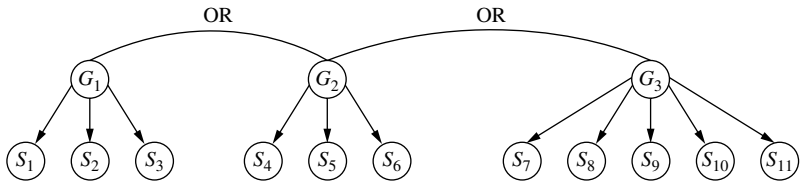


Figure 12.18 Multiple sensitization goal trees.

Suppose that registers R_1 and R_8 already had the values needed to sensitize the fault. Then it may be the case that by loading a new value in R_2, the fault would become fully sensitized. That is essentially what the MinGoal strategy attempted. If the goals that were already satisfied could hold their values, then the cost of justifying the one remaining goal, in this case register R_2, would usually be much less.

MaxGoal was the strategy originally implemented in TDX. Basically, it took all of the goals S_i and worked backwards, attempting to justify all of them. Quite often this was unnecessary, and sometimes even counterproductive, because the goals that were already satisfied, when backtracing, could introduce conflicts unnecessarily. The strategy that was eventually settled on was to use the MinGoal strategy, and if that failed, then a modified version of the MaxGoal strategy would be attempted. In this modified version, if the original MinGoal was satisfied and if one or more of the other goals became corrupted while justifying the MinGoal, then the process would repeat, but this time the MinGoal would be augmented with the goal(s) that became corrupted. Regardless of the strategy, the objective was to start at the goal state GS with a set of goals, as illustrated in Figure 12.19, and work back through one or more time frames, until finally the initial state IS is reached. Along the way, while backtracing, at different timeframes primary input values are assigned so that when proceeding forward from IS and making those assignments, the goal state is reached.

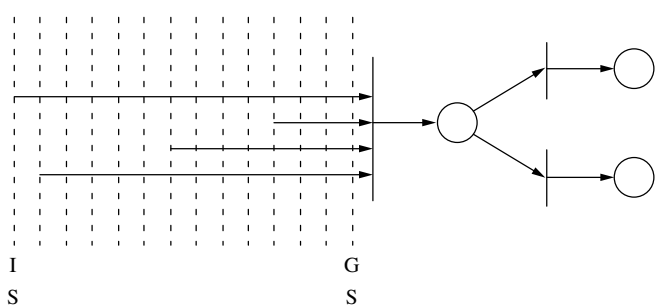


Figure 12.19 Petri net illustrating event alignment.

12.8.11 Functional Walk

Up to this point the discussion of TDX has focused on the fault-directed mode, but another of its strategies was a pure functional mode. The purpose of this mode was to explore the circuit and acquire information that could be used to build up a knowledge base. TDX was quite effective at identifying state machines. As TDX evolved to handle more complex circuits, including circuits with multiple, interacting state machines, counters, mode control registers, status registers, instruction registers, and data path elements, it was recognized that processing these individual constructs did not provide a sufficient understanding of how to process a complete circuit. The real challenge was to define and encapsulate in a knowledge base the relationships that represent, in some sense, an understanding of the interactions between the myriad functions.

It was possible to capture knowledge that characterized relationships between functions while building goal trees in the fault-directed mode. However, this approach tended to be fragmentary because goal trees were focused on building a bridge from current state to target state. So, rather than rely solely on information gleaned during fault-directed test, it was decided that a more rigorous approach was needed. The *functional walk* was developed for this purpose. In this mode of operation, TDX attempted to methodically exercise all of the functional units in the circuit. The first step, as in other modes of operation, was to initialize all of the sequential elements. Then, an initial pass, a so-called *static analysis*, was made by examining the circuit description in order to extract information about relationships between the functional units.

After the static analysis was completed and the initial knowledge base was constructed, functional walk commenced. This represented a *dynamic analysis* of the circuit. During this operation, TDX attempted to transition through all of the states of the state machines in the circuit, and it attempted to exercise all of the functional elements. In the process of walking through the various functions, a test vector file was created. The vectors that were created were annotated. Then, if the circuit exploration led to an unexpected destination (e.g., an infinite loop or a dead end from which the circuit could only recover via a circuit reset), the vectors could be examined to determine how that state was reached. Note, however, that although this was intended to be a functional analysis of the circuit, and every attempt was made to explore all explicit states of the state machines, the vectors were, nevertheless, treated as test vectors and fault-simulated. Often the fault coverage results obtained during functional walk yielded fault coverage results comparable to those obtained from design verification vectors provided by the circuit designers.

Although functional walk focused on state machines, it could explore other control structures and data path elements. For example, Section 7.8.2 gives examples of vectors generated algorithmically for various functions. These vectors can be used as the basis for exercising a data path function. For example, a behavioral vector could specify values on the A and B input ports of an ALU, as well as the carry-in and the mode control. Success then is defined as the ability to manipulate the circuit so as to apply those inputs and generate the correct result at the output of the ALU.

Another goal might be to force a counter to count. This might appear as a program counter incrementing—for example, performing a NOOP instruction.

The functional walk was implemented as a series of interconnected tables, such as illustrated in Section 2.9.2. An implementation based on ROBDDs, called FAME (fast, accurate, memory efficient), was in the planning stages. At the time when TDX was under development (late 1980s to early 1990s), much progress was being made in the field of BDDs and it was believed that these could help to solve one of the most critical problems in TDX, namely, small blocks of highly sequential state machines and other circuitry interconnected with counters and control registers. These often represented 5% of the circuit but 95% of the problems for TDX. The goal was to identify blocks of RTL code representing complex, convoluted functions, extract them, and represent them internally by means of ROBDDs rather than by interconnected tables, since BDDs would permit a uniform, methodical approach to finding a solution.

12.8.12 Learn Mode

In previous sections it was pointed out that humans impose a great deal of structure on a problem, based on their formal training and experience. Within that framework of knowledge and understanding, their so-called frame of reference, humans often understand the general concepts, but may not know all the details pertaining to how a particular instance of a device works. That is where trial and error begins. The human guesses at a solution and then attempts to ratify that guess. In digital logic this ratification, or verification, is usually accomplished by means of a simulator. If the simulation reveals that the desired goal was not achieved, then the human studies the results, performing, in effect, a postmortem, to determine what caused circuit behavior to deviate from intended behavior. This analysis often leads to an alternate plan of action, one that may involve tweaking the original plan, or, conversely, it may indicate that a radical rethinking of the problem is warranted at this point.

Using information to correct a course of action is quite commonplace. For example, when attempting to catch a ball, humans use visual feedback to guide the hand. An example of a negative feedback servomechanism is illustrated in Figure 12.20.[26] This figure may depict the action of a device such as a thermostat. The temperature θ_i in a room is monitored and compared to a desired temperature θ_0. If the room becomes too chilly, the difference $e = \theta_i - \theta_0$ becomes too large, and this error signal causes the furnace to be turned on. When the difference e between measured temperature and desired temperature becomes sufficiently small, the furnace is turned off. Obviously, in order to employ feedback, it is necessary to know the desired outcome.

The concept of feedback applies equally well in software. Consider the system depicted in Figure 12.21.[27] This *expert system* begins life with a knowledge base composed of facts and relationships, usually drawn out of an expert who devoted a lifetime to his or her craft. The facts are usually organized in the form of rules. As the expert system is put to use in real-life situations, oversights and flawed judgment are detected. These oversights may include missing, wrong, or incorrectly interpreted data. The links or relationships between the data may also be incorrect. When detected, these oversights are corrected, or, more precisely, the knowledge base is refined.

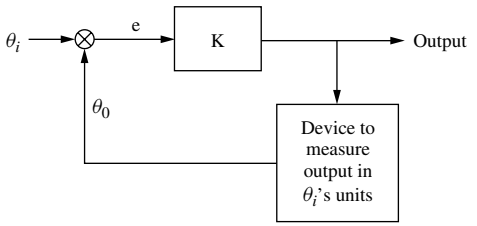


Figure 12.20 A negative feedback servomechanism.

A major difficulty with rule-based systems is that it is difficult to know when a set of rules is complete (or, put another way, when the specification is complete). In *PC Magazine*'s "abort, retry, fail?" column, it was reported that a German couple out for a Christmas drive ended up in a river—apparently because their luxury car's computer navigation system forgot to mention that they had to wait for a ferry. The driver kept going straight in the dark, expecting a bridge, and ended up in the water.[28]

The knowledge base in Figure 12.21 lists some of the sources for rules used in TDX. The behavioral and structural knowledge are obvious, coming from the RTL and gate-level circuit descriptions. Domain specific knowledge was derived from functions that were encountered while reading the circuit description. Characteristic features of these functions helped to identify them while reading in the circuit and, once it was determined, for example, that a particular construct was a counter, it was then known how it behaved (cf. Section 12.6.3, Library of Parameterized Modules).

User inputs can be used to further refine the knowledge base. If the user believes that TDX has incorrectly classified an object, he or she can override the classification attributed to the object by TDX. Furthermore, the user can add information to further refine the knowledge base. As an example, perhaps the user knows that the

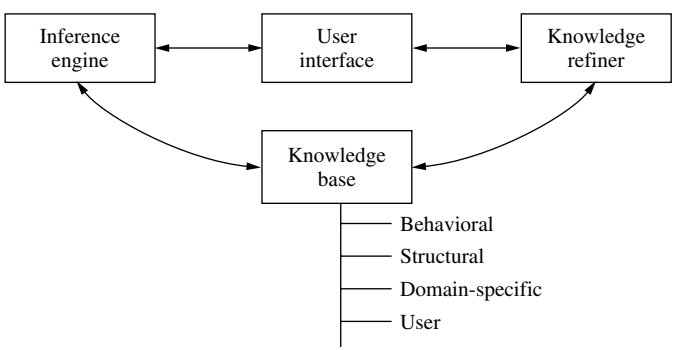


Figure 12.21 Expert system employing feedback.

circuit needs a complex initialization sequence. That information can be added to the knowledge base. The goal is to organize all of the available information in the most productive manner so as to make the user productive.

Despite the existence of the knowledge base, mistakes were still made when creating input sequences. Frequently, when generating a test vector sequence, unintended side effects would invalidate the sequence. These side effects often came from signals that had a large amount of fanout. An obvious example of this was a tendency to toggle set or clear lines in an attempt to load a register with all 0s or all 1s, but the problem was not confined to these two signals. Failure to carefully observe constraints (cf. Section 12.8.8) could also invalidate a sequence.

When a sequence failed to elicit the desired behavior from a circuit, the logic simulation capability of the fault simulator was employed in support of the *learn mode*. During this mode the system acts somewhat like the servomechanism or expert system. TDX knows the current state of the circuit, and it knows the goal state. Hence, if the circuit fails to reach the goal state, TDX has, in effect, an error signal that can be used to indicate the degree to which the circuit diverges from the target goal. As an example, assume that 22 clock cycles are generated in order to reach the goal state. During creation of this sequence, TDX has a record of all the intermediate states that must be reached. Suppose, now, that after the 10th clock cycle the simulator finds that the circuit diverges from the state predicted by TDX for that period.

At this point the so-called error signal is the degree to which the circuit deviates from the goal state. One or more of the storage elements reached a state other than that predicted by TDX. If the circuit reached the correct state on the previous clock cycle, TDX can further examine the circuit to determine if it is even possible to reach the goal state. By expanding signals in order to create trial sequences and simulating them, TDX can examine the results for all possible or reasonable signal values. The state of the circuit at this point may indicate where an applied signal caused the circuit to deviate from the intended state. If the cause of the incorrect transition is determined, then the knowledge base can be annotated to indicate that a particular state transition has some pitfalls that must be avoided.

Sometimes TDX is unable to repair a sequence. This might happen if it is not possible to get from the current state to the next state in a single clock period. It may be necessary to insert multiple states, or it may be necessary to back up one or more clock cycles. The simulator was instrumented so as to be able to roll back a simulation to the current state that existed when the vector sequence was created. From there, it could then resimulate the circuit for any specified number of clock cycles so as to allow further analysis of the state transitions that occur during a given clock period.

This entire learn mode operation is illustrated in Figure 12.22. If the search heuristics drive the circuit from the current state to the objective state, then the sequence that was generated is added to the end of the current test program. However, if the search heuristics fail to drive the circuit into the objective state, then an analysis is performed. The purpose of the analysis is to determine why the heuristics failed. The analyzer cannot, in itself, change the search software, but it can annotate the model so

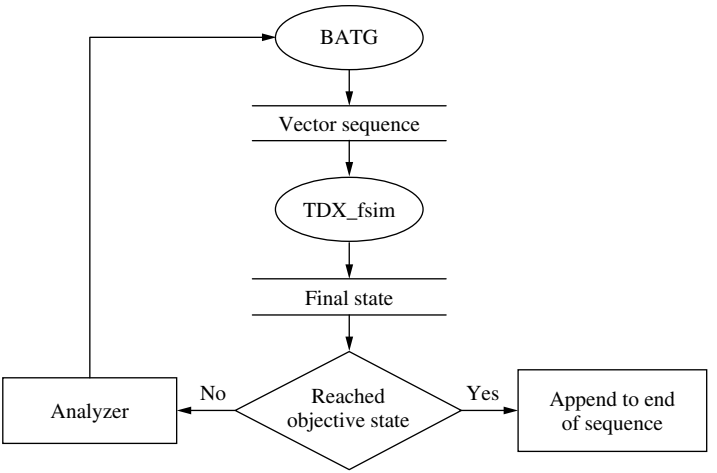


Figure 12.22 Learn mode.

as to influence the search heuristics. A key part of the analysis is the ability to roll back the state of the circuit to the state that existed before the current subsequence was applied. Then, after each vector is simulated, the state of the circuit is compared to that which was predicted by the search heuristics while it was generating the subsequence.

When the errant state is discovered, the analyzer can again back up, this time a single step, and attempt to regenerate the next step in the subsequence, but being careful not to revisit the previously visited state. It is possible that there is no identifiable path to the objective state from the current intermediate step, in which case the analyzer can again back up one step, assuming that it is not at the beginning of the subsequence.

12.8.13 DFT in TDX

During the period when TDX was in development (1988–1992), gate count in logic circuits was growing rapidly, and Moore's law was in full force. Entire PCBs were being subsumed into one or two ICs. In addition, memory elements and other custom-designed modules were beginning to appear on the same die as the random logic. Design-for-test was becoming a more important aspect of test because, even if it were possible to develop a test that could achieve high fault coverage, the test would require so much time on the tester that its cost would be prohibitive. As an example, consider the power management feature for green computers. Power management contains large counters that sometimes have little more than a reset and a clock to control their operation. If the counter reaches its maximum count, the computer is put into sleep mode. If a key on the keyboard is depressed, the counter is reset. Clearly, to test this structure, it has to be possible to either load or

scan-in selected values in order to set up initial conditions for individual targeted faults. Occasionally, when TDX was being benchmarked, customers would recognize that certain features, such as the sleep mode, were testability problems, and models would be scaled-down. Troublesome features were deleted from the model and, during test, input combinations were applied that would freeze these features in a known but inoperable state.

TDX imposed a limit on the number of time frames used to sensitize or propagate faults. This limit was user-controllable. Often when faults could not be sensitized or propagated within the default number of time frames, it was found that the number of time frames required was far greater than the default. These faults usually pointed to areas of the design where partial scan could significantly reduce the number of test vectors needed to generate a test for the fault. Despite the presence of other forms of controllability/observability (C/O) analysis, RTL-level analysis frequently proved to be a more reliable indicator. It exposed areas of a design where large numbers of faults could not be tested within a reasonable number of time frames, and it pointed directly at areas of a design where partial scan was needed.

Features of TDX that were particularly effective at identifying areas of a design in need of DFT support were the initialization phase and functional walk. In the initialization phase, each flip-flop and latch was established an a stand-alone goal, with the object being to initialize it and cause it to switch. Elements closest to the primary inputs were selected first, so that when it came time to initialize a flip-flop, those in its cone had already been processed. When an area of a design proved to be uninitializable, there was frequently a root cause. By carefully analyzing the problem at the RTL level, TDX could often identify a source that not only was uninitializable, but also controlled other functional areas of the design. One option was to retry initialization of the root cause with a higher threshold on the number of allowable time frames. The theory being that if the root cause could be initialized, then those functions controlled by the root cause could become controllable. Another strategy was to force initialization of an element that appeared to be a root cause, and then re-try initialization of those elements that depend on a known value in the flip-flop designated as a root cause. Functional walk was also useful in this regard. By working forward in time, it would often find solutions that TDX could not find by backtracing from each storage element that was established as an individual goal.

The use of RTL to guide the search for test sequences often had the effect of significantly reducing the length of a test program. Whereas a gate-level ATPG employing partial scan might require 80–90% partial scan to reach a stated fault coverage, TDX often could reach that fault coverage goal in fewer vectors and with considerably fewer scan elements, usually requiring as little as 10–25% partial scan. When targeting large combinational blocks, such as multipliers and floating point arrays, a partial scan in which some control elements are scanned while the data path is exercised from the primary inputs may have the dual advantage of fewer vectors while simultaneously providing some of the benefits of a behavioral test.[29]

Section 7.8 examined strategies for creating test vectors directed at basic functions, such as counters, ALUs, multiplexers, and so on. TDX had the ability to target these constructs at the RTL level. After a function had been targeted, precomputed sequences could be used to efficiently test those functions. These vectors had the advantage that they not only were efficient, but they could also test for fault models such as bridging faults, which might not be detected using patterns generated at the gate level by a gate-level ATPG. If fault simulation revealed that there were undetected faults remaining after the precomputed sequence of vectors had been applied, then DEPOT could go in and generate test vectors for the remaining undetected faults.

12.9 DESIGN VERIFICATION

In the early days of digital IC design, manufacturing test and design verification shared many common tools and methods. Simulation was the workhorse of both test and verification. A concurrent fault simulator, at its core, was basically a logic simulator with specialized processing added in order to permit two or more nearly identical circuits to be simulated concurrently. Even ATPG, at some conceptual level, is merely an extension of what the designer does when verifying a circuit; that is, a function in a circuit is identified and an algorithm is invoked whose purpose is to create a sequence of vectors that exercise the logic. TDX extended that concept in order to imitate and take advantage of some of the less-algorithmic, more ad hoc techniques practiced by humans.

The emergence of designs ranging in size from hundreds of thousands to millions of logic gate equivalents has necessitated a fresh look at approaches to design and test. A quarter century of trying to create ATPG programs capable of dealing with sequential circuits has proven unproductive. Even if such programs existed and could generate manufacturing test sequences that provide high coverage, the test sequences required to detect structural faults would generally be far too lengthy to be economically practical. DFT has to be part of the solution for almost all designs.

For design verification a similar situation exists. It is not possible to simulate all possible combinations of inputs and latch/flip-flop states. Consider a 1,000,000 gate-equivalent circuit. Typically, one could expect between 5% and 10% of those gates to be flip-flops. But, assume instead a conservative 1%, or 10,000 storage elements. Also assume 400 primary inputs. Then, there are $2^{10,000+400}$ unique combinations on the combined state elements plus inputs (cf. Problem 3.3). This is illustrated in Figure 12.23. The graph conveys a sense of the magnitude of the design verification task. For a given number of input vectors, as circuit size increases, the percentage of the design that is evaluated by the vectors decreases rapidly. As circuits grow larger, verification involves selecting and simulating a subset of the possible stimuli based on an understanding of intended circuit behavior. If the wrong subset is chosen or an inadequate subset is chosen, critically important interactions in the circuit fail to be examined, with the result that errors may exist in the design when tape-out occurs.

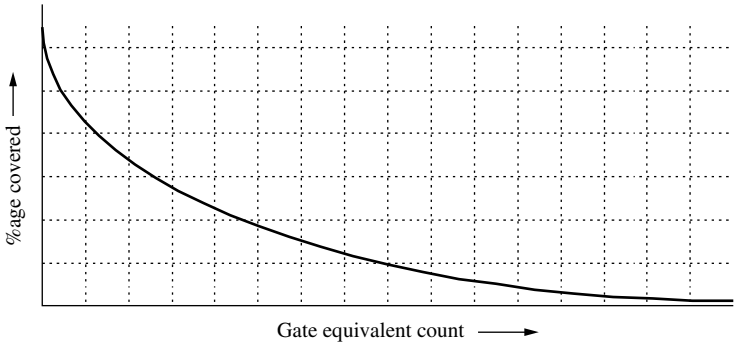


Figure 12.23 Combinatorial explosion.

12.9.1 Formal Verification

Clearly, it is not possible, with the size of today's circuits, to exercise all combinations of values on inputs and state elements. This has led to a growing interest in formal verification. Four approaches to formal verification have received considerable attention from researchers:

Theorem proving
Equivalence checking
Model checking
Symbolic simulation

12.9.2 Theorem Proving

This method of proving that a design is correct depends on logic. Thus, before we discuss this method, we introduce some basic definitions. In logic, a *statement* is a sentence or phrase that affirms or denies an attribute about one or more objects. A *proposition* is a declarative statement; it affirms or denies an attribute about one or more objects, but it is either true or false. A statement can be ambiguous or open to debate. For example, the statement "that was a good movie" may be true for one viewer but false for another viewer. However, the statement "the movie ran for more than two hours" is either true or false and can be verified. Hence it is a proposition.

An *atomic proposition* is a basic proposition, one that cannot be broken down into two or more smaller units. A *compound proposition* is one that is composed of two or more atomic propositions that are connected by logical connectives, such as AND, OR, NOT, XOR, equivalence, and implies. When discussing propositional logic, it is customary to represent these operations by the symbols $\wedge$, $\vee$, $\neg$, $\oplus$, $\equiv$, and $\rightarrow$, respectively. We will adhere to these conventions in this section. Some additional comments are in order regarding propositional logic. First, equivalence is what we often think of as the exclusive-NOR operation; that is, the expression $A \oplus B$ is true if

A and B are different. But $A \equiv B$ is true if A and B are the same. Also, $A \rightarrow B$ is equivalent to $\overline{A} \vee B$; that is, the expression is true if A is false, or if A is true and B is true. The expression $A \rightarrow B$ is only false if A is true and B is false, which can be interpreted to mean that a true premise cannot imply a false conclusion.

We have used propositional logic throughout the text; it is the backbone of the digital industry. However, it has its limitations, which in turn has led to extensions. *Predicate logic* is one such extension. A predicate of a proposition is that which is affirmed or denied of the subject. For example, in the sentence "the dog is in the house" the predicate is the word "in". In predicate logic the sentence is usually constructed using prefix notation. The aforementioned sentence would be written as IN(dog, house) in prefix notation. Using infix notation, the sentence would be written "dog IN house," although the infix form is more commonly used in conjunction with Boolean and mathematics operators, as in the expression $A + B$, where the plus sign (+) is the predicate.

Predicate logic is used in conjunction with *theorem proving*. While a complete discourse on theorem proving is beyond the scope of this text, an example with a simple circuit can help to illustrate the concept.

Example Using predicate logic, the circuit in Figure 12.24 is described in terms of its intended behavior by means of the following equation:

$$\text{Net\_Spec}(A,B,C,Z) = \neg(\, A \wedge B \,) \wedge C = Z$$

In the following equation the circuit is described in terms of its implementation:

$$\text{Net\_Impl}(A,B,C,P,Q,Z) = (\, A \wedge B = P) \wedge (\neg P = Q) \wedge (\, Q \wedge C = Z)$$

The object is to eliminate the intermediate variables in the predicate Net_Impl so that it resembles the predicate Net_Spec. Given that $\neg P = Q$, the variable Q can be replaced by $\neg P$ wherever it appears. The middle term then becomes an identity, so it can be eliminated. This yields

$$\text{Net\_Impl}(A,B,C,P,Q,Z) = (\, A \wedge B = P) \wedge (\, \neg P \wedge C = Z)$$

Now, using the substitution $P = A \wedge B$, the expression for Net_Spec results. ■■

Theorem-proving software programs have been used to advantage on large combinational blocks of logic, but they require considerable manual guidance; hence much research is required before they can be used on an everyday basis by anyone but experts.

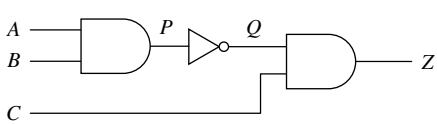


Figure 12.24 Net implementation.

12.9.3 Equivalence Checking

Equivalence checking attempts to demonstrate that two circuits produce equivalent (but not necessarily identical) behavior. A typical application would be a comparison of an RTL description to a synthesized version of that same circuit. The synthesized circuit may include gate-level functions designed using a schematic editor, functions pulled off a library, and functions purchased from a third party, or it may be a combination of logic derived from all these sources. The gate-level circuit may include scan circuits or may be otherwise modified (e.g., retiming, which shifts logic from one side of a flip-flop to the other) in order to reduce path delays.

Demonstrating equivalence of the two circuits can be accomplished as described in Section 2.11. Recall that ROBDDs are unique; hence if two circuits are represented by identical ROBDDs, then the circuits are identical. Given two ROBDDs, the Traverse algorithm, described in Section 2.11.2, can be used to compare corresponding vertices of the two ROBDDs. It is also possible, given circuits f and g, to perform Apply($\oplus$, B_f, B_g), where B_f and B_g are the ROBDDs for the functions f and g.

The comparison of two circuits f and g is accomplished by comparing cones of combinational logic bounded by primary outputs and internal flip-flops. This can be seen in Figure 12.25 (cf. also Figure 7.21). The combinational cone driving flip-flop DFF*i* has inputs *A*, *Bd*, *C*, and *Dd*. Inputs *A* and *C* are primary inputs, whereas *Bd* and *Dd*, the delayed *B* and *D* signals, are driven by flip-flops DFF1 and DFF2. When comparing two circuits at different levels of abstraction (e.g., an RTL description and a gate-level description), it can be difficult to correlate the storage elements in the two models. Recall, from Section 12.8, that TDX employed a map file to link flip-flops in the gate-level model with their counterparts in the RTL model. In equivalence checkers this correlation is done by means of a computer program, and the primary outputs and flip-flops are referred to as *state points*.[30] The first step in mapping cones is to find the cones for each state point in each of the two models being compared. The inputs driving a cone will be state points. Corresponding cones for the two models will have identical state points as inputs.

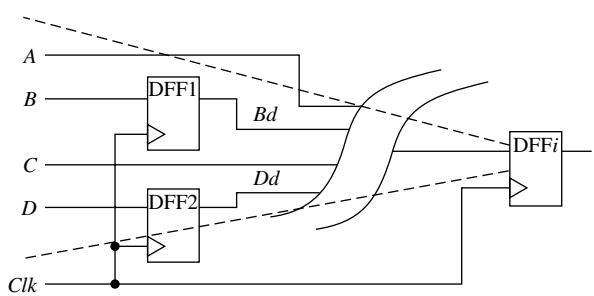


Figure 12.25 The combinational cone.

Given the logic corresponding to the cone for either the RTL or gate level, a ROBDD can be generated for each cone using the Apply algorithm described in Section 2.11. The Traverse procedure can then be used to compare cones of corresponding state points. An alternative is to generate logic equations for the cones. Given the netlist, an equation similar to Net_Impl is generated. This equation is expressed in terms of the inputs to the cone, as well as internal signals. The equation is then reduced. This can be done as in the previous section on theorem proving, where it was shown that Net_Spec and Net_Impl were functionally identical blocks of logic. The advantage is that storing the circuit description as equations may require much less memory than storing the ROBDDs. When creating the equation for a cone, the cone can first be rank-ordered. Then, rather than build up the complete netlist and eliminate the internal variables after the complete cone netlist has been constructed, elimination of internal variables can be performed incrementally as the cone equations are being developed, just as is done when creating ROBDDs.

When the equation for a cone in a netlist has been created and is expressed completely in terms of state points, it remains to demonstrate that it matches the equation for the corresponding cone in the RTL circuit. One way this can be accomplished is by numbering the state points; that is, given n state points, assign each state point a unique variable x_i, for $1 \leq i \leq n$, as is done for ROBDDs. Then convert the equations to disjunctive normal form and order the terms based on their subscripts.

Example Given: the equation

$$(x_1 \cdot x_2 + \bar{x}_2 \cdot x_3) \cdot (x_1 \cdot \bar{x}_4 + x_3 \cdot x_4)$$

This can be translated to

$$x_1 \cdot x_2 \cdot \bar{x}_4 + \bar{x}_2 \cdot x_3 \cdot x_4 + x_1 \cdot \bar{x}_2 \cdot x_3 \cdot \bar{x}_4 + x_1 \cdot x_2 \cdot x_3 \cdot x_4$$

Terms with three variables appear first, and they are sorted in ascending order based on their subscripts. If terms occur with identical variables, sort them based on their binary equivalents; that is, assign the value 1 to those in true form and assign 0 to those that are negated. Equations for corresponding cones can then be compared on a term-by-term basis to determine if the cones are identical. ■■

Equivalence checking has reached a level of maturity where it is commonly used as a routine part of regression testing. Whenever changes are made to a netlist—for example, when logic is added after synthesis, including scan, clock trees, changes (tweaks) to improve speed, clock-retiming, and so on—it is standard practice to run regression tests to ensure that the RTL and synthesized versions remain functionally equivalent. It must be pointed out that equivalence checking targets implementation errors, not design errors. If an error exists in the RTL, an equivalent structural model will contain that same error.

12.9.4 Model Checking

It is generally accepted that the ideal verification test is an exhaustive test—that is, one that explores all possibilities and either confirms that a design responds correctly, or provides an *error trace* showing where the design responds incorrectly. We have seen that for simulation, a complete, exhaustive evaluation is not possible, even for modest-sized circuits. But *model checking* does perform exhaustive checking. A model checker, depicted in Figure 12.26, is a software package that accepts as inputs a circuit model and a set of properties.[31] The model may be expressed in Verilog, VHDL, or some similar such hardware design language. The model checker evaluates the properties against the model. If the model does not satisfy the properties, then the model checker provides a sequence, sometimes called a *witness*, that leads from a start state to a conflict state; otherwise the model checker confirms that the model satisfies the property.[32] The model checker uses ROBDDs for internal representation of the circuit. This data structure can be quite efficient, permitting the model checker to exhaustively check much larger circuits than would be possible if data structures and linked lists were used. Previously, when we discussed the Test Design Expert (TDX), we pointed out that the model checking paradigm was being considered for incorporation into the TDX framework (Section 12.8.11). Unfortunately, the TDX project ran short of funding before the plans could be realized.

The properties that are submitted to the model checker are expressed in a propositional temporal logic, called *computation tree logic* (*CTL*). CTL recognizes the logic operators introduced in the previous section on theorem proving, but extends their reach by introducing additional operators, called *temporal operators*, that are able to specify temporal relationships.[33]The temporal operators do not recognize specific time increments, but instead express temporal relationships about variables in a circuit. For example, if a *request* is issued, will a *grant* eventually be received? The operators express properties of a CTL formula f along a *temporal sequence*, or *computation path P*, which is the sequence of states visited during an execution sequence. The temporal operators and their meanings are as follows:

- **G** *globally*, a formula is true along a path P now and at all future times.
- **F** *eventually*, a formula will be true along path P at some future time.
- **X** *next*, a formula is true at the next instant of time.
- **U** *until*, the expression f **U** g, is true along a path p if g is true in some state s, and if f is true in all preceding states.

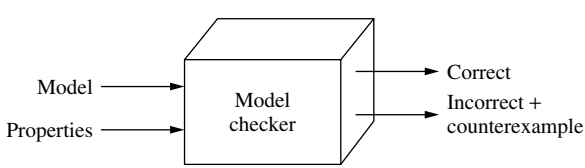


Figure 12.26 Model checker flow.

A formula that is true eventually may in fact be true at present, so a proposition or formula that is globally true is also eventually true, but not vice versa. The next operator is generally understood to refer to an environment, such as a synchronous circuit, where the next instant of time refers to the next active clock edge.

CTL is a *branching time logic*; that is, from any given state, there may be several possible branches (next states). *Path quantifiers* are used to indicate whether a formula f is true along some, or all, possible branches from a state s. These are as follows:

A *universal path quantifier*, f holds for all possible branches from state s.
E *existential path quantifier*, f holds on at least one path from state s.

The path quantifiers are used in conjunction with the temporal operators. For example, the CTL formula **EG***f* is used to state that there exists a path in which formula f is always (globally) true. The universal and existential quantifiers can be expressed in terms of one another. The formula **EG***f* is equivalent to $\neg$**AF**$\neg f$. Verbally, this states that "there exists a path along which formula f is globally true" is equivalent to "there is no path along which formula f is eventually false."

Two important properties that are often submitted to model checkers are *safety properties* and *liveness properties*. A safety property expresses some undesirable behavior that a circuit must avoid, while a liveness property expresses some desirable behavior required from the circuit. To illustrate this, consider a *traffic light controller (TLC)* that regulates the flow of traffic. It must not permit simultaneous traffic in north–south and east–west directions. This is a safety requirement. On the other hand, cars reaching the intersection from all directions must eventually be serviced by the TLC. This is a liveness requirement. The properties for the TLC can be written independently of the TLC circuit model. In this respect, the properties serve as a (partial) specification of the design. They are totally independent of the implementation; every TLC must satisfy these safety and liveness properties, regardless of whether the TLC is regulating traffic in a busy intersection on Main Street or on a highway where a traffic light senses occasional traffic from a farm road and eventually gives that traffic the right-of-way.

Example The following is a sample of the properties that must be satisfied for an implementation of the traffic intersection in Figure 12.27.

AG($\neg$N–Go $\wedge$ N–s $\rightarrow$ **AF** N–Go)
AG($\neg$S–Go $\wedge$ S–s $\rightarrow$ **AF** S–Go)
AG($\neg$W–Go $\wedge$ W–s $\rightarrow$ **AF** W–Go)
AG($\neg$E–Go $\wedge$ E–s $\rightarrow$ **AF** E–Go)
AG$\neg$(E–Go $\wedge$ (N–Go $\vee$ S–Go))
EF(N–Go $\wedge$ S–Go)

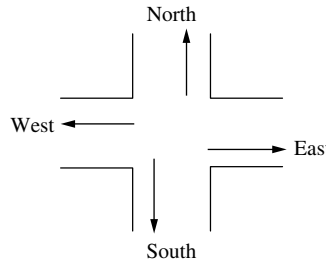


Figure 12.27 Traffic light controller.

In this example, the letters N–s, S–s, E–s, and W–s denote sensors. The letters N–Go, S–Go, E–Go, and W–Go are traffic signals for the indicated directions. So, in reading the first property, the letters **AG** indicate that the property in parentheses must hold globally throughout the design. The property in parentheses states that if the sensor for north-going traffic is active, but the traffic signal for traffic going north is not active, then eventually, on all paths, the N–Go signal must become true. This is a liveness property. The next three properties are interpreted similarly.

The fifth property is a safety property. It states that the traffic signal must never be active in the east direction while it is also active in the north or south direction. The last property ensures simultaneous north/south traffic flow. This is optional; the specification may actually call for north-going traffic to simultaneously have the right-of-way for a left turn. In that case, south-going traffic must be inhibited. ■■

Given a specification, the model checker processes it piecemeal. In order to do this, the CTL formula is parsed into atomic propositions and operators. Consider the first CTL formula in the example. The parse tree for this formula is shown in Figure 12.28. Parsing is performed on the basis of operator precedence. Note first that the terminal vertices are atomic propositions. For the logic operators, the unary $\neg$ has a higher precedence than the binary operators, and the logic operators $\wedge$ and $\vee$ have higher priority than $\rightarrow$. Where there is any doubt about how a formula might be parsed, it is recommended that parentheses be used to remove the ambiguity.

The parsed CTL equation is evaluated by the model checker, one vertex at a time, starting with terminal vertices. The model is examined to determine if there are any states that satisfy N–Go. Then it is necessary to find states that satisfy $\neg$N–Go—that is, states that do not satisfy N–Go. Next, states satisfying N–s must be identified. The sets of states that satisfy $\neg$N–Go and N–s are intersected to satisfy the subformula $exp_1 = \neg$N–Go $\wedge$ N–s. For the subformula $exp_2 =$ **AF** N–Go the model checker starts with the states in which N–Go is true, then backs up from these states, looking for states such that all paths from those states eventually satisfy N–Go.

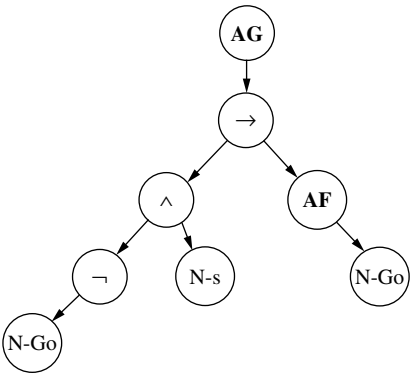


Figure 12.28 Parse tree for CTL formula.

Now, given the sets of states satisfying exp_1 and exp_2, it becomes necessary to satisfy the implication ($\rightarrow$). Recall that implication is equivalent to $\overline{exp}_1 \vee exp_2$. Given the set of states that satisfy exp_1, the set of states that satisfy $\overline{exp}_1$ is $S - exp_1$, where S is the set of all states (the universe). The last step is to determine if the formula $\mathbf{AG}(\overline{exp}_1 \vee exp_2)$ is true. It is true if the expression $\overline{exp}_1 \vee exp_2$ is true in all states.

We introduce another example to illustrate model checking. This example, illustrated in Figure 12.29, is called the *mutual exclusion problem*,[34] often referred to as *mutex*. It is a nondeterministic state transition graph; the transitions are shown, but the conditions under which they choose one transition over another are not shown. For the purposes of model checking, it is not important how a particular state was reached; it is only important to know if the state can be reached. The circuit represented by the figure consists of two concurrent processes, P_1 and P_2. Each process can be in one of three regions (a region can be a software or hardware function of arbitrary size):

N_i	noncritical region
T_i	trying region
C_i	critical region

The processes can be software functions or hardware functions. A software process may be trying to access a file on a hard drive. A hardware process could be one of several I/O devices trying to write to memory. In either case, software or hardware, only one process is permitted to access the resource at any given time. When a process has no need to access the resource, it is in its noncritical region. When it needs the resource, it enters its trying region. Eventually it is granted access to the resource, at which point the process enters its critical region.

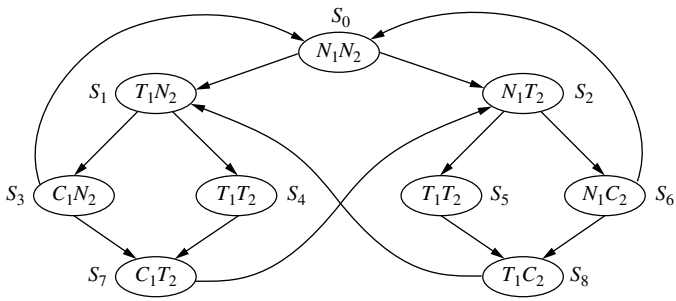


Figure 12.29 Mutual exclusion system.

It is assumed that a process remains in its critical region for only a finite duration of time. Hence, each process gets as many accesses to the resource as it wants. However, since the processes are not permitted to simultaneously access the resource, the state C_1C_2 is not permitted. Note, however, that S_4 and S_5 both represent T_1T_2. It would seem that these states could be merged. However, if that were done, the merged state, call it S_4', would have transitions to C_1T_2 (S_7) and T_1C_2 (S_8). There would then be a loop S_1–S_4'–S_8 (actually an infinite path when unrolled). Note that the graph in Figure 12.29 is nondeterministic; hence if the arbiter were biased as a result of some implementation detail, the system could remain in that loop indefinitely, with the result that C_1 would never get access to the resource. In technical terms, the system with the merged state S_4' would violate a CTL liveness formula $\mathbf{AG}(T_1 \rightarrow \mathbf{AF}C_1)$.

It can be shown that the circuit in Figure 12.29 satisfies the CTL formula $\mathbf{AG}(T_1 \rightarrow \mathbf{AF}C_1)$. This is demonstrated using the equivalent expression $\neg T_1 \vee \mathbf{AF}C_1$. The model checker must identify those states for which either T_1 is not true or C_1 is eventually true ($\mathbf{AF}C_1$). The set of states that satisfy $\neg T_1$ is $\{S_0, S_2, S_3, S_6, S_7\}$. The set of states that satisfy $\mathbf{AF}C_1$ is found iteratively, using the following function:

```
function MC_AF(C_1)
{
  X = 0;
  Y = MC(C_1)
  while (X ! = Y)
    {
      X = Y;
      Y = Y ∪ {s | s → s′ ∈ Y};
    }
  return(Y);
}
```

$\mathrm{MC}(C_1)$ denotes the model checker searching for the set Y of states that satisfy C_1. Once that set is found, the function $\mathrm{MC}_{\mathrm{AF}}(C_1)$ loops through the set of states,

searching for states whose branches all transition to C_1 or $\mathbf{AF}C_1$. It is readily confirmed that initially Y = $\{S_3, S_7\}$. Then, after the first pass Y = $\{S_3, S_7, S_1, S_4\}$. After the second pass Y = $\{S_3, S_7, S_1, S_4, S_8\}$, and after the third pass Y = $\{S_3, S_7, S_1, S_4, S_8, S_5\}$. There is one more iteration, but no new states are found, so the function halts. The set Y is termed a *fixed point*. In general, if F(Y) = Y, then the set Y is termed a fixed point. The set of states S_0, S_2, and S_6 form a loop that does not satisfy $\mathbf{AF}C_1$. However, they satisfy $\neg T_1$, so the union of the two sets of states equals the entire set of states. In addition to checking properties, the algorithms can also perform *reachability analysis*, wherein the program determines which states can, and which cannot, be reached from an initial state.

It was pointed out previously that advances in model checking have been aided by advances in ROBDDs, in large part due to the fact that ROBDDs can represent circuits in a very compact form. As was pointed out above, the CTL formula is parsed and processed piecemeal. The temporal operators represent the greatest challenge. We saw that the CTL formula **AF** had to be solved iteratively. There are eight pairs of quantifier/temporal operator combinations. Fortunately, they can all be expressed in terms of **AF**, **EU**, and **EX**. However, expressing these as ROBDDs requires a considerable amount of additional machinery, and it would take us too far afield to delve more deeply into the subject. Several good texts have been written on the subject. The first-time student may find the text by Huth and Ryan[35] to be a good introduction to the subject. For the reader looking for a more rigorous discussion, the texts by Clarke et al.[36] and Kropf[37] can be found helpful.

We now explain how ROBDDs support model checking. However, a more modest example will be used to explain the sequence of operations. The circuit in Figure 12.30 is a simple 2-bit counter, with its state transition graph shown to the right. We will create a ROBDD representing the state transitions for this circuit. The state transition equations for this circuit are listed in Figure 12.30.

A ROBDD for a sequential circuit can be created from a truth table representing the state transitions. This truth table is given in Figure 12.31. In that table, M denotes the state machine transitions. Where $M = 1$ there is a transition; for example, there is a transition from $X_1, X_2 = 0, 0$ to $X_1', X_2' = 1, 0$. Note that the variables are interleaved; that is, it might be expected that the variables would be listed in the

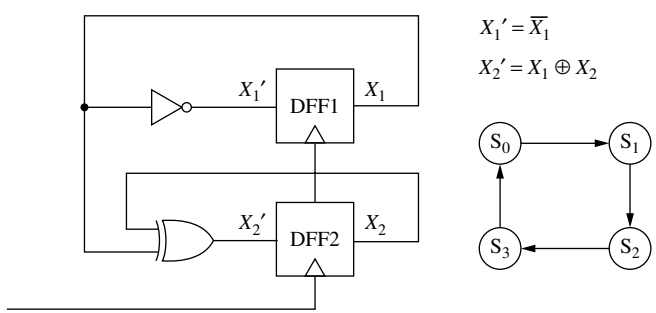


Figure 12.30 Two-bit counter.

X_1	X_1'	X_2	X_2'	M
0	0	0	0	0
0	0	0	1	0
0	0	1	0	0
0	0	1	1	0
0	1	0	0	1
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	0
1	0	0	1	1
1	0	1	0	1
1	0	1	1	0
1	1	0	0	0
1	1	0	1	0
1	1	1	0	0
1	1	1	1	0

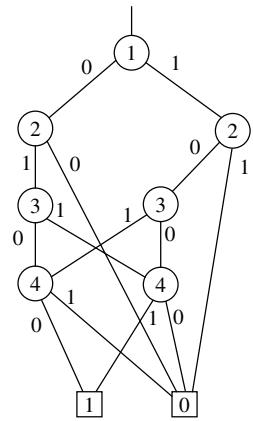


Figure 12.31 Truth table for two-bit counter.

order $X_1\ X_2\ X_1'\ X_2'$, but experience suggests that interleaving usually produces more compact ROBDDs. Also note that there are four entries for which $M = 1$; these entries correspond to the four transitions of the state transition graph. The ROBDD is illustrated to the right of the truth table, where X_1, X_1', X_2, X_2' are mapped to indices 1, 2, 3, and 4, respectively.

Once the truth table has been created, generating the ROBDD is straightforward (cf. Section 2.11). Four paths in the ROBDD lead to the terminal vertex with value 1. A problem with this method is that it relied upon the truth table. The truth table is helpful for describing, conceptually, what is being done, but for large circuits the truth table takes more memory space than can be afforded. To avoid the expense of building the truth table, the Apply operation is used to build the ROBDD directly.

The transition relation for a synchronous circuit can be expressed as $\prod_{1 \le i \le n} (f_i = X_i')$. For the 2-bit counter $n = 2$. Expressed in this form, the equations for the 2-bit counter become $(X_1' = \overline{X}_1) \cdot (X_2' = X_1 \oplus X_2)$. The equal sign (=) is treated as an exclusive-NOR, so the equation becomes $(X_1' \overline{\oplus}\ \overline{X}_1) \cdot (X_2' \overline{\oplus}\ X_1 \oplus X_2)$, which can be simplified to $(X_1' \oplus X_1) \cdot (\overline{X}_2' \oplus X_1 \oplus X_2)$. Repeated applications of the Apply and Reduce algorithms can now be used on this equation to create the ROBDD directly, without the intermediate step of creating a truth table.

Once the ROBDD has been built for the model, the next step is to build a ROBDD for the CTL formula. As was done previously, the initial step is to parse the CTL formula. The next step is to find the states in which the atomic propositions hold. To illustrate this, we assume that the states X_1 and X_2 are atomic propositions and associate the states S_i with (X_1, X_2) by mapping the states into their binary

equivalents: $S_0 \rightarrow (0, 0)$, $S_1 \rightarrow (1, 0)$, $S_2 \rightarrow (0, 1)$, $S_3 \rightarrow (1, 1)$. Then, to find the states that satisfy atomic proposition X_1 we perform Apply($\cdot$, B_M, B_{X1}).

The model checker is most effective on *reactive circuits*—that is, circuits that react to stimuli from their environment. The mutex circuit described above is an example of such a circuit. Bus controllers, cache coherency circuits, and instruction pipelines are other examples of circuits that have benefited from the use of model checkers. These types of circuits are characterized by the fact that many events occur concurrently, and checking out all the possible combinations of events by means of simulation is impractical. As a result, subtle design errors may escape detection until after an IC has been put into service. All appears well until, suddenly, a user configures the IC in a way that had not previously been anticipated.

Several examples have been cited in the literature where model checking has identified problems in designs thought to have been correctly designed. A cache coherency protocol for a distributed, shared memory multiprocessor was verified by means of model checking.[38] Subtle errors were found that had extremely low probability of occurrence in simulation runs. The IEEE Futurebus+ cache coherence protocol is another example of a complex system where model checking was able to find subtle bugs in the design.[39] An IBM design team used model checking to verify the control logic in a circuit they were designing.[40] Their records showed that formal verification (FV) detected 24% of the errors that were discovered, even though simulation had preceded the use of FV. Of the errors detected by FV, they believed that 40% of those errors might not have been detected by simulation.

Despite the efficiency with which ROBDDs can represent data structures, model checking still has problems coping with large circuits. "Even simple cores like a PCI interface are simply too big for model-checking tools, unless someone exercises a great deal of cleverness to figure out how to abstract the design and boil it down to something that's small enough for a model-checking tool to deal with."[41] Because model checking is exhaustive, every variable doubles the state-space that must be explored, subjecting it to state explosion.

It is estimated that model checking can handle circuits containing up to 200–400 latches or flip-flops, depending on the amount of memory available on the host computer. Clearly, this is not adequate to handle multi-million gate circuits. However, the parts of a circuit that present problems, because of concurrency, can often be extracted from the larger circuit and verified standalone, as was done by the IBM design team. The standalone environment is probably the most severe test, since there are no restraints on the design in that environment. Once the verification is complete, the function can be placed in a library where it can be called into applications as needed. In general, some of the things that can be done to handle large circuits include:

Composition	Break a problem into smaller units.
Reduce data paths	For example, model an ALU as a 2-bit data path.
Abstraction	Eliminate variables not needed to prove a property (every variable doubles memory requirements).

Induction	If an arbiter must handle n units, perform model checking first on a model with two units, then three, and so on.

Other approaches include combining model checking with random simulation. When the model checker fails to verify a property because it runs out of memory, the system can attempt to verify a property by resorting to random simulation. It has also been suggested that the ultimate verifier will combine model checking with theorem proving.[36] The Accellera standards committee has endorsed IBM's *Sugar* as a standard to drive assertion-based verification.[42] This language is a superset of CTL in which many constructs have been added to make the language user-friendly, without necessarily extending its expressiveness.

Symbolic Model Verifier (SMV) and *Verification Interacting with Synthesis (VIS)* are two popular university-developed model checkers. SMV, developed at Carnegie Mellon University, has its own language, whereas VIS, developed initially at the University of California, Berkeley, and subsequently in collaboration with the University of Colorado, has a utility called *Verilog to MV (VL2MV)* that reads Verilog and compiles it to BLIF-MV, the native language of VIS.

12.9.5 Symbolic Simulation

Symbolic simulation resembles logic simulation in many respects. Stimuli are presented to a simulator, which then applies the stimuli to a circuit model in order to predict its response. The major difference between the symbolic simulator and the logic simulator is that the set of stimuli accepted by the symbolic simulator includes symbolic values in addition to binary values. The simulator response appears at the output in the form of expressions. This is illustrated in Figure 12.32. In this five-input circuit, there are 32 possible binary input combinations. To exhaustively check all combinations applied to this circuit would require simulating 32 possible combinations. By using symbols and examining the expression at the output, all 32 possible combinations are evaluated simultaneously.

For a five-input circuit, with its 32 input combinations, symbolic simulation does not offer a significant gain, but consider the case of a 32-bit ALU. It simply is not feasible to simulate all possible binary combinations (cf. Problem 3.3). While symbolically simulating a single vector along a data path takes significantly more CPU

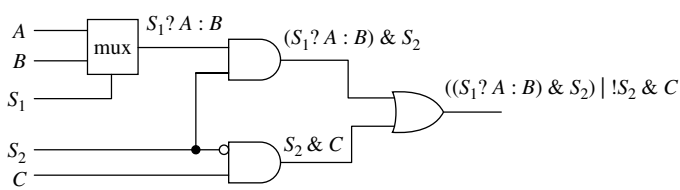


Figure 12.32 Symbolic simulation.

time than simulating a single vector composed of binary bits, symbolically simulating a single vector will be faster than simulating tens of thousands of randomly generated binary vectors. As to how to decide where to employ symbolic simulation, note that an approach often used in testbenches during verification is to set up a loop in which binary stimuli are randomly generated and simulated until there is a high degree of confidence that the design is free of errors (Section 12.5). The presence of loops and random stimuli in the testbench is often a clue as to where to employ symbolic simulation.

When simulating an array with predictable behavior, such as an ALU, analysis of results may be easier if binary values are assigned to the control bits. For example, suppose an ALU has four control bits, thus being able to perform any of 16 possible arithmetic and logic operations. In such a case, each of the 16 operations could be simulated with symbolic values applied to the data path inputs, and each of these 16 could be individually examined to determine whether the response is correct. If one of the 16 operations is the AND operation, then the expected output would be $a_0 \cdot b_0$, $a_1 \cdot b_1$, ..., $a_{31} \cdot b_{31}$. This flexibility, being able to specify symbolic values on some inputs and specify binary values on others, is particularly useful if there is insufficient computer memory to represent all the inputs symbolically.

One of the benefits of symbolic simulation is its ability to handle large circuits. It may be somewhat constrained by the need for storage to contain the symbolic equations, but this has only negligible impact on its capacity. Perhaps of more concern is the amount of time and effort required to examine simulation response, and the possibility that examining complex responses may be an error-prone operation, given that equations are expressed symbolically. Also to be considered is the time needed to examine response, versus the time it takes to set up random simulations and create a self-checking testbench configuration.

Equivalence checking was discussed in Section 12.9.3. When state points match, corresponding cones can be checked without too much difficulty. However, two designs may possess equivalent behavior, but they may be so different internally that it is difficult or impossible to get a match for the state points. For example, one design may be expressed in terms of muxed state machines, while state machines in the other design may be one-hot encoded. Since symbolic simulation only concerns itself with values applied to the inputs, as well as with responses that appear at the outputs, it may be able to help determine whether or not the two designs are behaviorally equivalent. If a mismatch occurs on the outputs, it is possible to pin down the internal node where the behaviors diverge, recognizing, of course, that internal representations will not be exact if state points do not exactly match.

Another argument put forth for symbolic simulation is the ease with which a simulation environment can be converted to accomodate symbolic simulation. There is almost no learning curve, since the HDL model and testbench that were used for logic simulation are easily adapted to perform symbolic simulation. This stands in contrast to theorem proving and model checking where a major learning curve is required.

For data path logic, the number of cycles of simulation required to complete the simulation will be the number of cycles required to propagate the values from the

inputs, through the circuit, to some observation point, such as an output port or an internal register. The data path may be an ALU, as mentioned earlier, or the data path may be an internal memory or a data switching device such as a multiplexer. Internal delays can also be represented symbolically, and they can be accumulated as the signals propagate forward to an output.

12.10 SUMMARY

Design verification is the process of demonstrating that a design satisfies a specification, which in turn must satisfy a set of requirements imposed by an end-user. The importance of a complete, accurate, and unambiguous specification tends to be overlooked. In software projects it is reported that 40–60% of all errors are requirements errors. A *requirement* is a capability or feature needed by an end-user to solve a problem or achieve an objective. Requirements management is a systematic approach to identifying, organizing, communicating, and managing the requirements of an application.[43] With respect to requirements, it has been said[44] that "the really serious mistakes occur in the first day." Furthermore, the longer the time lag from specification to discovery and repair of a bug (requirements error), the more expensive the cost of the fix. This closely parallels the rule-of-10 (cf. Section 1.8) of hardware design, which asserts that the cost of finding and fixing a bug in hardware increases by a factor of 10 at each level of integration. Clearly, if the specification is incorrect or incomplete, then verifying that the implementation matches the specification simply verifies that the correct bugs have been designed into the product.

In the early days of digital circuit design, it was possible to simulate all or nearly all possible input combinations to a circuit and develop complete confidence that the design would work as intended. That is no longer possible. A typical digital circuit ranges in size from hundreds of thousands to millions of logic gates. It is also likely to contain many embedded memories, small and large. Some of the modules are designed in-house, others are purchased on the open market. Those that are purchased on the open market may be soft IP (intellectual property), meaning they are described at the RTL level, or they may be hard IP, in which the end-user gets a file containing a description expressed in a graphical description language, but the user gets little or no information describing the internal workings of the IC.

The specification for the design may be written in a formal language, or it may be written in everyday English (or some other spoken language). The spoken language tends to be imprecise, with considerable ambiguity, redundancy, confusion, and/or incompleteness. Sometimes project managers are in a hurry to start a project, and they may initiate the project with an incomplete specification, on the expectation that the specification will be completed as the project goes forward. But it has been pointed out that requirements errors are 10 times more costly to correct than other kinds of bugs.[43] This is because a change in requirements may necessitate a wholesale restructuring of the design, whether it be software or hardware. So proceeding with a partially complete specification may result in a large part of that development effort being discarded, at a considerable cost in time and money.

Contemporary logic designs are created by teams of logic designers, each of whom has his or her preferences when it comes to such things as coding style, logic partitioning, and verification practices. A large, expensive design, if successful, may live for many years and may be the basis for many generations of follow-on products. The design must be consistent throughout in order that new team members can take over and enhance or add new features to the design, or port it to a new technology with minimum confusion.

The choice of verification methods is one of the issues that team members must decide at the start of a design project. Many verification methodologies are available, and the choice of which method(s) to use may, in part, be determined by the nature of the product being designed. Simulation can handle designs of any size, but the amount of time available for simulation limits the number of cases that can be explored. Random stimulus generation does not solve this problem, but distributes stimuli more evenly across the design so that obscure bugs are more likely to be found. It is also less labor-intensive than targeted stimulus generation, so a user may spend more time simulating and analyzing the results of simulation. A hardware accelerator would be especially useful in a random stimulus approach since, if no bugs are found, the user is ready to simulate a new set of stimuli almost as soon as the currently running simulation has completed. One drawback to random stimuli is the fact that, if a bug is encountered, it may take longer to isolate the problem, since it is not always immediately clear what part of the circuit the stimuli were targeting.

Formal methods are attractive for targeting logic in reactive systems. However, they are generally not capable of assimilating entire designs. Those areas of control logic that are complex and obscure, and thus possible sources of subtle bugs, can be tested in a stand-alone mode, independent from the rest of the circuit. Then, once the design team is satisfied that the function is free of bugs, it can be put in a library and made available to anyone who has a need for it. In fact, some IEEE protocols have been verified using formal verification, with the result that some bugs overlooked by many members of the standards committee were found and corrected.

TDX was a commercial effort that made good progress but eventually ran out of money. TDX used conventional arrays and linked lists to represent and process circuit images in memory. It might have benefited from the adoption of ROBDDs, as well as the use of other model checking methodologies. It also was limited by the capabilities of its resources. Workstations were one to two orders of magnitude less powerful in both memory and CPU speed.

Because model checkers exhaustively analyze a circuit model, they are limited by the amount of available memory, and thus are most useful in circuits that are small but complex, such as arbiters and bus controllers. Model checkers might benefit from the addition of methods such as those developed for TDX. The thoroughness of model checkers is an advantage when analyzing extremely difficult functions such as arbiters, but there are areas of a design, particularly combinational blocks of logic such as those found in the data flow section, where theorem proving is more effective. Once the block of logic has been demonstrated to be correct, it can be placed in a library and henceforth it is only necessary to confirm that it has been correctly interfaced to the circuit in which it will be used. The entire area of

formal verification is an active area of research, and it will continue to be so well into the future.

PROBLEMS

12.1 List all the possible functional errors (error seeding) that you can think of that can be applied to the b16ctr model in Section 12.6.3.

12.2 Repeat the previous problem for the decoder in Section 12.6.3.

12.3 Using the state-machine description (Figure 12.5) of the SM8 circuit in Figure 8.44, find a test sequence for an SA0 on the input of gate 16 driven by gate 11. Contrast this with the effort required to find a test relying solely on the gate-level description.

12.4 Starting at the reset state, S_0, provide a list of the state transitions that exercise every arc in the state machine of Figure 12.5. If you have access to a fault simulator, fault simulate this sequence on its gate equivalent, Figure 8.44, to determine if it will detect every fault in the circuit. (Note that in this small circuit, if you do not have access to a fault simulator, faults can be injected and simulated serially by means of a logic simulator).

12.5 In Section 2.12 there is a Verilog RTL model for the circuit in Figure 2.43. Write a set of test vectors that exercise the RTL model. If you have access to a fault simulator, evaluate your vectors against the gate-level model.

12.6 Assume that the counter in Section 12.6.4 does not have a parallel load or reset, but does have a left shift. Assume also that the counter is 32 bits in width. Write the sequence required to load the counter with all 0s.

12.7 For the sensitization tree of Figure 12.7, assume that the state machine is currently in state S_7. Draw a sensitization tree that identifies paths from S_7 to S_4. Explain your rationale for each of the possible paths from S_7 to S_4, and why you might choose each of them.

12.8 Again using the sensitization tree of Figure 12.7, how many unique paths can you find from state S_4 to S_1 without using the reset?

12.9 Create a Petri net representation for the JK flip-flop.

12.10 Create a Petri net representation of the state machine in Figure 12.16(b).

12.11 In the Petri net of Figure 12.8, a token in P_2 causes tokens to appear in P_3 and P_4. However, a token in P_6 can only cause a token to appear in either of P_2 or P_7, but not both. Explain this.

12.12 Identify the places in Figure 12.10. That is, for each P_i, define the register that needs a specific value and determine what that value is. Complete the Petri net in Figure 12.10. With a complete graph, you must be able to identify several sequences that take the circuit from the reset state to the Goal state.

12.13 Using the *M* and *T* state machines illustrated in Figure 12.16, draw a product state machine that has a corresponding state for every state pair in the original *M* and *T* state circuits.

12.14 Using the product state machine created in the previous problem, create a search tree that will cause the circuit to transition from the reset state to state $<M_3, T_4>$.

12.15 Write the equations for a 3-bit counter like the 2-bit counter in Figure 12.30. Create a truth table like the one in Figure 12.31.

12.16 Prove that the muxed and one-hot encoded versions of the circuit in Figure 9.30 are equivalent.

12.17 Using the inheritance properties of C++, develop a series of structures that characterize a Counter in the LPM library so that a user can specify the properties of the counter to be used in his or her design and be able to select the necessary structure from a library of C++ structures.

12.18 You are performing black-box testing of software whose job is to read three numbers from a file and determine whether those three numbers define a scalene, isoceles, equilateral, or right triangle. How many tests do you need?

REFERENCES

1. Quinnel, R. A., Kill Bugs Early with Software-Test Tools, *EDN Magazine*, May 23, 1996.
2. Debany, W. H. et al., Design Verification Using Logic Tests, *Proc. IEEE Int. Workshop on Rapid System Prototyping*, June 1991, pp. 17–24.
3. Baird, Mike, Designers May Find C++ to Their Liking, *ISD Magazine*, October 2001, pp. 48–53.
4. Thomas, D. E. et al., A Model and Methodology for Hardware-Software Codesign, *IEEE Des. & Test*, September 1993, pp. 6–15.
5. Kalavade, A., and E. A. Lee, A Hardware-Software Codesign Methodology for DSP Applications, *IEEE Des. Test*, September 1993, pp. 16–28.
6. Syzgenda, S. A., and A. A. Lekkos, Integrated Techniques for Functional and Gate-Level Digital Logic Simulation, *Proc. 10th Design Automation Conf.*, 1973, pp. 159–172.
7. Thomas, J. J., Common Misconceptions in Digital Test Generation, *Comput. Des.*, January 1977, pp. 89–94.
8. Bening, L., and H. Foster, *Principles of Verifiable RTL Design*, Kluwer, Boston, 2000.
9. Beizer, Boris, *Black-Box Testing*, John Wiley & Sons, New York, 1995, p. 9.
10. Bellon, C. et al., Automatic Generation of Microprocessor Test Programs, *Proc. 19th Des. Autom. Conf.*, 1982, pp. 566–573.
11. Aharon, A. et al., Verification of the IBM RISC System/6000 by a Dynamic Biased Pseudo-Random Test Program Generator, *IBM Syst. J.*, Vol. 30, No. 4, 1991, pp. 527–538.
12. Wood, David A. et al., Verifying a Multiprocessor Cache Controller Using Random Test Generation, *IEEE Des. Test*, Vol. 7, No. 4, August 1990, pp. 13–25.

13. http://www.edif.org/lpmweb/intro/functions.html
14. Breuer, M. A., and A. D. Friedman, Functional Level Primitives in Test Generation, *IEEE Trans. Comput*., Vol. C-29, No. 3, March 1980, pp. 223–235.
15. Levendel, Y. H., and P. R. Menon, Test Generation Algorithms for Computer Hardware Description Languages, *IEEE Trans. Comput*., Vol. C-31, No. 7, July 1982, pp. 577–587.
16. Belt, John E., *An Heuristic Search Approach to Test Sequence Generation for AHPL Described Synchronous Sequential Circuits*, Ph.D. dissertation, University of Arizona, 1973.
17. Hill, F., and B. M. Huey, A Design Language Based Approach to Test Sequence Generation, *IEEE Comput*., Vol. 10, No. 6, June 1977, pp. 28–33.
18. Azema, P. et al., Petri Nets as a Common Tool for Design Verification and Hardware Simulation, *Proc. 13th D.A. Conf.*, 1976, pp. 109–116.
19. Dennis, J. B. et al., Computational Structures, *Project MAC Progress Report VIII*, July 1971, pp. 11–52.
20. Torku, E. K., and B. M. Huey, Petri Net Based Search Directing Heuristics for Test Generation, *Proc. 20th Des. Autom. Conf.*, 1983, pp. 323–330.
21. Torku, Emmanuel K., *Fault Test Generation for Sequential Circuits: A Search Directing Heuristic*, Ph.D. dissertation, University of Oklahoma, Norman, OK, 1979.
22. Hill, F. J., and G. R. Peterson, *Computer Aided Logical Design with Emphasis on VLSI,* 4th ed., Appendix B, John Wiley & Sons, New York, 1993.
23. Freundlich, Y., Knowledge Bases and Databases, *IEEE Comput*., Vol. 23, No. 11, November 1990, pp. 51–57.
24. Maxwell, Peter C., Reductions in Quality Caused by Uneven Fault Coverage of Different Areas of an Integrated Circuit, *IEEE Trans. CAD*, Vol. 14, No. 5, May 1995, pp. 603–607.
25. Huey, Ben M., *Search Directing Heuristics for the Sequential Circuit Test Search System (SCIRTSS)*, Ph.D. dissertation, University of Arizona, 1975.
26. Arbib, M. A., *Brains, Machines and Mathematics*, "Cybernetics," Chapter 4, McGraw-Hill, New York, 1964.
27. Bharath, R., *An Introduction to Prolog*, Appendix B, "Artificial Intelligence, Expert Systems and Prolog," TAB Books, 1986.
28. *PC Magazine*, Vol. 18, No. 10, May 25, 1999, p. 274.
29. Maxwell, P. C., R. C. Aitken, V. Johansen, and I. Chiang, The Effectiveness of I_{DDQ}, Functional and Scan Tests: How Many Fault Coverages Do We Need?, *Proc. Int. Test Conf.*, October 1992, pp. 168–177.
30. Blackett, R. K., As Good as Gold, *IEEE Spectrum*, Vol. 33, No. 6, June 1996, pp. 68–71.
31. Clarke, E. M., and R. P. Kurshan, Computer-aided Verification, *IEEE Spectrum*, June 1996, Vol. 33, No. 6, pp. 61–67.
32. Clarke, E. M. et al., Efficient Generation of Counterexamples and Witnesses in Symbolic Model Checking, *Proc. DA Conf.*, 1995, pp. 427–432.
33. Bochmann, Gregor V., Hardware Specification with Temporal Logic: An Example, *IEEE Trans. Comput*., Vol. c-31, No. 3, March 1982, pp. 223–231.
34. Clarke, E. M. et al., Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications, *ACM Trans. Programming Languages and Systems*, Vol. 8, No. 2, April 1986, pp. 244–263.

35. Huth, R. A. M., and M. D. Ryan, *Logic in Computer Science: Modelling and Reasoning About Systems*, Cambridge University Press, Cambridge, England, 2000.
36. Clarke, E. M., O. Grumberg, and D. A. Peled, *Model Checking*, MIT Press, Cambridge, MA, 1999.
37. Kropf, T., *Introduction to Formal Hardware Verification*, Springer, Berlin, 1999.
38. McMillan, K. L., *Symbolic Model Checking: An Approach to the State Explosion Problem*, Ph.D. dissertation, Carnegie Mellon University, Pittsburgh, PA, 1992.
39. Clarke, E. M. et al., Verification of the Futurebus + Cache Coherence Protocol, in *Formal Methods in System Design*, Kluwer, Boston, MA, 1995, pp. 217–232.
40. Schlipf, T. et al., Formal Verification Made Easy, IBM J. Res. Dev., Vol. 41, No. 4/5, 1997.
41. Dill, David, quoted in Formal Verification: Assessing a Critical Technology, panel discussion, Barbara Tuck, moderator, Computer Design, June 1998.
42. http://www.eedesign.com/story/OEG20020509s0075
43. Davis, Alan M., and Dean A. Leffingwell, Using Requirements Management to Speed Delivery of Higher Quality Applications, Technical Report 0001, Requisite, Inc., Boulder, CO, 1996.
44. Rechtin, Eberhardt, The Synthesis of Complex Systems, *IEEE Spectrum*, July 1997, pp. 51–55.

INDEX

Symbols

@ symbol 292
Numerics
0-1 transition 523
0-controllability 397–398
1-0 transition 523
1-controllability 397, 399, 401
74181 ALU
  Verilog model 359–360
  gate-level drawing 376
95% confidence level 347
9-value ITG 246–249, 278

A

A-algorithm (AALG) 184–188
  back propagation 185–188
  chain segment 187–188
  DRBACK 187–188
  DROPIT 187–188
  flexible signal(s) 185–187
Abbreviated descriptor cell 145, 149
ABIST (array BIST) 479
ABT test 477
AC test 298
Acceptable fault coverage 471
Acceptable quality level (AQL) 2, 17, 23, 348, 388, 551
Accumulate signatures 460
Accuracy, definition of 285
Action/check pair 587
Active fault tolerance 495, 499
Activity vector 180–181
Acyclic circuit 40, 92, 260, 263
Adaptive experiments 266
Addition of two vectors 538
Address decoder faults 522
Address lines 515
Addressable Registers 411–412, 415
Ad-hoc DFT solutions 388–389, 393–395
Ad-hoc techniques 635
AHPL (A Hardware Programming Language) 35, 597–598, 601, 604
Algorithmic test 356–361
Aliasing 470–471
Almost-full-scan 427
Alpha particles 537
Ambiguity
  simulation 272
  state machine 267–269, 272–275
Ambiguous region 63
AmZ8065 Burst Error Processor 502
Analysis of a Faulted Circuit 122–125
AQL *see* Acceptable quality level
Artificial intelligence (AI) 599
Assertion checker 577
Asynchronous circuit 40, 61
ATLAS (Abbreviated Test Language for All Systems) 320
Atomic proposition(s) 636, 642, 646
ATPG/fault simulator link 378–379
ATPG/user controls 380
Automated optical inspection (AOI) 317
Automated X-ray inspection (AXI) 317–318
Availability of a system 503, 506

B

Backdriving 309
Backtrace 186, 190–202, 205, 208, 210, 618–620, 622

Digital Logic Testing and Simulation, *Second Edition*, by Alexander Miczo
ISBN 0-471-43995-9 Copyright © 2003 John Wiley & Sons, Inc.

Backtrack 186, 193, 199, 202, 218
Balanced acyclic sequential circuit 262–264
Ball grid array (BGA) 317, 433
Bare-board testing 302
Bare die 24
Basis of *V* 539
Bed-of-nails 302, 307–308, 313
Behavioral fault modeling 353–356
Behavioral fault simulation 361–364, 575–576
Behavioral intermediate form (BIF) 608
Behavioral MUX 354–356
Behavioral processing algorithms 593–596
Behaviorally equivalent circuit 338
Biasing functions 585
Bidirectional goal search 624–625
Bidirectional pins 3344–345
Bidirectional signal flow 75
Binary decision diagram(s) (BDD) 86–101, 219–224, 615, 630, 638–639, 645–647
  0-experiment 223
  1-experiment 223
  apply algorithm 92, 96–100, 220, 638–639, 646–647
  class 1 faults 221
  class 2 faults 221
  composition algorithm 101
  faulting the BDD Graph 220–224
  reduce algorithm 91–96, 646
  restrict algorithm 101
  total ordering on the variables 91
Binary message stream 459
Binary operators 642
Binning 3
Binomial expansion 464, 544
Binomial probability distribution 12
BIST *see* built-in self-test
Bit line 63–64
Bit-changer 467
Black box testing 357, 418, 488–489, 569
Block coverage 365, 576
Block oriented analysis 108–110
Blocking assignments 580
Blocking signal 38, 139
Blocking value 127
Boolean difference(s) 210–216, 615
Boolean satisfiability 216–219
Bound nets 195
Branch address hashing 497
Branch coverage 576
Branch node 496
Branch-and-bound 189–190
Branching time logic 641
Breadth-first search 624
Break all cycles 432
Bridging faults 335, 337, 552
BSDL (boundary scan description language) 442
Building goal trees 617–618, 626
Built-in logic block observer (BILBO) 463–464, 486, 532
Built-in-self-test (BIST)
  benefits of BIST 452
  self-Test for BIST 531
  self-test SRL 475
  Self-test control macro (STCM) 479
  self-testing circuit 498
Bulletins, scheduling 65
Burn-in 4–5, 559
Burn-in failures 560
Burst error correction 499–503
Bus contention 61, 129, 554
Bus functional model (BFM) 344
Bus keeper 553
Bypassing memory 418

C

Canonical form 266
Capacitive coupling 562
Capture internal state at registers 581
Capture line 210
Capturing design verification vectors 344–346
Case statement 590
Catastrophic failure 552
Causal link 147
Causative links 74
CCITT-16 polynomial 483
Cell libraries 296
Central processing unit (CPU) 489, 513
Chain rule 215
Channel connected component 79, 83, 340
Characterize new devices 296
Characterized by a number of parameters 590
Charge sharing 75
Charge, trapped 62
Checking sequence 270
Checkpoint arcs 332

Checkpoint faults 331–333, 364
Circuit Initialization 349–350
Circuit level model 37
Circuit partitioning 137, 465–466
Clock rate tester 286
Clock skew 415, 423
Cluster parameter 14
CMOS *see* Complementary metal oxide semiconductor
Code coverage 365–367, 575–578, 585
Code coverage versus fault simulation 366
Code inspections 581
Coefficient field 457
Coincidental correctness 576
Column decoder 515
Combinational controllability 397–398
Combinational feedback loop 417
Combinatorial explosion 127, 165, 522, 599
Common ambiguity 73
Commutative linear algebra 457
Commutative ring 456
Compiled code 570
Compiled simulator 44–48
Complementary Metal Oxide Semiconductor (CMOS) 38–39, 124, 338–339, 551
  stuck-open faults 454
Component evaluation 80
Component interface 487
Composite signal 1/0 167
Compound proposition 636
Comprehension Versus Resolution 371
Computation path 640
Computation tree logic (CTL) 640–646
  parsing the CTL formula 642–643
Computer Description Language (CDL) 35
Concurrent engineering 29
Concurrent fault simulation 139–149, 334, 362, 608, 616, 635
Concurrent operation 64
Concurrent processes 643
Cone(s) of logic 45–46, 105, 137, 252, 316, 374, 464, 481, 571, 638–639
Conflict(s) 172, 177–179, 186, 207, 258, 417, 624
Conformal coating 310
Conjunction 38
Conjunctive normal form (CNF) 39, 216–218
Connection function 76
Connectivity tables 406
Consistency operation 168
Consistent singular cubes 172
Constraint propagation 609, 625–626
Continuous loop, run in 480
Contrapositive 202
Control concurrency 574
Control faults 355
Control registers 568
Controllability 388, 394, 466, 551, 576–578
Controllability Equations 396–398
Controllability relation 490
Controllability/observability (C/O) 396–406, 609, 634
Controlling signal 108
Converged 143–144
Converging lists scheduler 65–66
COP (controllability and observability program) 403
Core limited die 407, 546
Core module(s) 8, 35, 158, 451, 567
Correct destination state 616
Correlate abstract states 601
Correlate environmental conditions 486
Cost of ownership 319
Cost to test a memory chip 521
Cost/benefit analysis 483
Counter, generic model 594
Cover 172
Cover line 210
Cover of F 172
Coverage Evaluation 575–578
Co-verification 573–575
Covering problem 494
Creation of effective stimuli 120
Criteria for selecting stimuli 345
Critical value on a node 206
Critical path tracing (CPT) 208–210
Critical path(s) 148, 205–207, 249–250
Critical race 50, 239
Critical region 643
Critical value(s) 205, 208
Cross-coupled latch 40–41
Cross-coupled NAND latch 234
Crosspoint fault 337
Crosstalk 286
Cube theory 171–182
  singular 172
  test cube 180–182

Current objective(s) 196, 198–199
Current time frame (CTF) 251–252
Cut feedback lines 49, 242–244
Cycle simulation 101–106, 571, 579
Cycle-free sequential circuit 431
Cyclic sequential circuit 40
Cyclic redundancy check (CRC) instruction 486

D

D flip-flop (DFF) 42–43, 89–90
DALG-II 369
D-algorithm 170–184, 272–273, 598
Data collection 478
Data faults 355
Data management system 10
DATA probe 474
Data retention test 533
Data sheet 568
Data transfer 495
Data width 591
DC test 298
D-chain 180, 183
Dead-end fault 369
Dead-end(s) 610–612, 618, 629
Deadening 371
Decision table 168
Declarative statement 636
Deductive fault simulation 151–152
Deep submicron (DSM) 26–27, 147, 552, 562
Defect 3
Defect level (DL) 15, 131
Defect size 27
Defects per million (DPM) 19, 21, 521, 531, 560–561
Defects per unit area 12
Defects that have strong nonlinear characteristics 557
Delay 50
  ambiguity 60
  calculations 70, 106–110
  distributed 40
  fault model 147–148, 333–334, 464, 562
  inertial 60, 67
  lumped 40, 52
  maximum 60
  media 60
  minimum 60
  nominal 55, 59
  transport 60
  turn off 60
  typical 60
  unit 55, 58–59
  zero 56–57, 570–571
DEPOT (DEductive, Path-Oriented Trace) 610, 614–619
Deracing 72
Derived clocks 416
Descripter cell(s) 67–68, 134, 588
Desensitizing 371
Design error coverage 579
Design error injection 581
Design Error Modeling 578–581
Design process 6– 7, 9
Design validation 568
Design verification 568–569, 579–580, 609, 635–636
Design verification testbench 366
Design verification vectors 364
Design-for-test (DFT) 20, 327, 341, 387–389, 412, 564, 633–634
Desktop Management Interface (DMI) 487–488
Desktop Management Task Force (DMTF) 453, 487
Destination goals 624
Destructive readout (DRO) 515
Detect electronics 311–312
Detectability of a fault 405
Detectable 167
Determining which vectors to retain 346
Device-under-test (DUT) 3, 120, 122, 284
D-frontier 180–181, 183–184, 191, 194, 199, 203–204
DFT *See* Design-for-test
Diagnosis 306
Diagnostic analysis 478
Diagnostic capability 134
Diagnostic control unit 80
Diagnostic program 7
Diagnostic program in ROM 473
Diagnostic Programs 497
Diagnostic software 485
Difference operator 212–213, 215
Differential fault simulation (DSIM) 149–151
Digital Description Language (DDL) 35

Digital sampling oscilloscope (DSO) 318
Dimension of *V* 539
D-intersection 180
Directed arc 495
Directly observable 613
Discrete increments 426
Disjunctive normal form 39, 639
Distinguishing sequence(s) 267–271
Distributed fault simulation 348
Diverged 143
Divide functionality (H/W, S/W) 570
D-list 183–184
D-notation 171
Domain specific knowledge 631
Dominant logic value (DLV) 208
Dominate(s) 130, 203
Dominator 203–204
Domino technique 297
Dot product 457
Double latch design 412–413
DPM *See* Defects per million
Drivers in parallel 392
Drop-in function 35
Dual Clock Serial Scan 410–411
Dual in-line packages (DIPs) 307, 432
Dump file 330
Duration of a strobe measurement 291
DUT *See* Device under test
Dynamic analysis 569, 629
Dynamic fault imaging, (DFI) 300–301
Dynamic memory 75
Dynamic partitioning 79
Dynamic RAM (DRAM) 64, 515–516
Dynamic tester 286–288

E

Early life failures 483, 559
E-beam probe 4, 299–301
ECC encoder circuit 542
Economical set of goals 618
Economics Of Test 20–23, 283
Edge propagation 148
Edge triggered flip-flops 42
EEPROMs (Electrically Erasable PROMs) 515–516
Effectiveness of Fault Simulation 23–24
Effectiveness of test stimuli 5–6
Effects of Memory 234
Electromigration 562
Electronic design automation (EDA) 9–11, 86, 296, 298
Electronic Design Interchange Format (EDIF) 589
Electronic knife 315
Elementary gate 335
Elementary in variable *x* 335
Embedded memories 524
Emission list (ELIST) 145
Emitter-coupled logic (ECL) 340, 484
Endmodule 329
Engineering change order (ECO) 34
Engineering Design System (EDS) 478
Engineering test station 296
Entropy *H* (bits per symbol) 537
EPROMs (Erasable PROMs) 515–516
Equal parity cover line 210
Equivalence checking 568, 636, 638–639, 649
Equivalence class of faults 129–131, 136, 331, 342, 478
Equivalent circuit(s) 340–341
Error 3
Error correcting codes (ECC) 29, 499–503, 537–543
Error detection and correction (EDAC) 188, 452, 486, 496, 499, 537–545
Error detection circuitry 499
Error patterns 462
Error seeding 579
Error signal 142, 486
Error trace 640
Errors to inject 580
Estimate of fault coverage 362
Euclidean division algorithm 457, 500–503
Evaluation techniques 70–71
Even parity check 542
Event 55, 64, 139
Event driven simulation 44, 54–56, 571
Event monitor 577–578
Event notice 64
Event propagation 106
Excess current 552
Excitation function 76
Excitation states 76
Exercise sequence 253
Exercise_part 293
Exhaustive *n−1* level search 599
Exhaustive test 464

Expected coverage $E(C)$ 465
Expected response 3, 284
Expected results 584
Expected signature(s) 453, 473
Expert system 630–632
Exploiting knowledge 587
Exploiting behavior 587
Expression coverage 365, 576–577
Extended backtrace (EBT) 250–252
Extended D-cubes 252–254
Extender cards 486
Extensibility 288
Extension field 458
Extension language 573
Extremal 172–173, 175

F

Failure analysis 300
Failure rate 543
Fall time 60
False negatives 320
False path 110
False reject rate 317
FAN (fanout oriented test generation algorithm) 193–202
Fanout branches (FOB) 209
Fanout free region (FFR) 195, 209, 331–332
Fanout point 195
Fan-out point objectives (FPO) 195–196, 199–202
Fast plunge 371
Fault Behavior for CMOS NOR 338–339
Fault collapsing 131
Fault coverage 14, 131, 134, 465
Fault coverage Profile(s) 350–351, 367
Fault coverage versus defect levels 17
Fault cubes 300–301
Fault diagnosis 132
Fault dictionaries 316, 351–352
Fault directed testing 356
Fault dominance 130, 136, 331
Fault dropping 137, 352–353
Fault effect(s) (FE) 142–147, 172, 209–210, 373, 405, 602, 610–611, 616
Fault file 326
Fault injection 134
Fault insertion in functional models 362
Fault list 169
Fault list collapsing 577
Fault list compiler 609
Fault-List Management 381
Fault list manager 609
Fault-List Partitioning 347
Fault models 127–129, 331–340
Fault origin 143–145
Fault partition sizes 347
Fault sampling 346–347, 609
Fault secure 498
Fault simulate RTL modules 362
Fault Simulator 311–312, 341–353, 616–617
Fault site event sources 150
Fault tolerance 495–505
Fault-directed mode 629
Fault-directed vectors 324
Fault-list compiler 326
Faults for functional primitives 356
Fault-secure 498
Feasibility studies 570
Feedback lines 39, 48
Feed-forward sequential circuit 427, 431
Fence multiplexer 481
FFR *See* Fanout free region
Field faults 337
Field reject rate 15
Field replaceable unit (FRU) 29
Field testing 453
Field-effect transistor (FET) 560
FIFO (first-in, first-out) memory 514
Fire code(s) 499
First silicon debug 479
First-degree hardcore 490
Fixed point of a set of states 645
Flattened netlist 326
Flush test 415, 477
Forced transition 523
Forcing values 205
Formal DFT 389
Formal verification (FV) 647
Formatting electronics 311
Forward implication 203
Free nets 195
Free run mode 472–473
Freeze pin 380
Frequency divider(s) 44, 390–392, 396
Functional board tester (FBT) 306, 315
Functional corners 567
Functional faults 355

Functional model 36
Functional test pattern generation algebra 595
Functional tester(s) 284, 287, 301, 303, 310–311
Functional walk 609, 629, 630
Functionally equivalent faults 522
FUNTAP (functional testability analysis program) 404

G

Galois field GF 456–458
Gate arrays 59
Gate equivalent, NAND 39, 74
Gate-level model 601
Gate-oxide short (GOS) 559
Gaussian distribution 13
General purpose tester 284–285
Generator matrix G 542
Generator matrix of V 540
Generic BIST circuit 525
Generic view of a function 588
Geometrical level model 37
Glitch 50, 52, 67
Goal ordering 622
Goal state 624
Goal tree(s) 605–606, 609, 618–620, 624, 629
Goals, competing 624
Go-Nogo test 3
Granularity 125–126, 337, 362, 381
Graph, definition of 86–87
  0-edge 87, 106
  1-edge 87, 106
  binary tree 87
  bipartite, directed graph 602
  directed acyclic graph (DAG) 87
  function graph 92
  graph methods for functional test 494–495
  isomorphic function graphs 92
  leaf vertex 87
  nonterminal vertices 87
  ordered tree 87
  parent of 87
  subgraph(s) 88, 92
  terminal vertex 87–88, 90, 92, 95–100
Graphical user interface (GUI) 487
Ground field 458
Group 455–456
  Abelian (commutative) group 456
  multiplicative group 458
Guard bands 296, 298
Guard circuit 308
Guidance file 380
Guided probe 313–316, 474

H

Hamming code(s) 538, 540–543
Hamming distance 104, 156, 540
Hamming weight 540
Handshaking protocols 567
Hard detect 129
Hard errors, logging 545
Hard-core IP 299, 650
Hard-core cell 420
Hardware accelerators 157
Hardware design language(s) (HDL) 7, 120, 325
Hazard(s) 50–54, 132, 239, 312
  0-hazard 51
  1-hazard 51
  detection 57–58
  dynamic hazard(s) 51, 57, 379–380
  function hazard 51
  logic hazard 51
  static hazard(s) 50, 57, 379–380
Hazard detection 52–54, 57, 58
Head lines 195
Head objective(s) 196, 198–199
Heuristic(s) 599, 601, 607, 612–614, 615, 622, 624
High frequency (HF) set 431
High leakage current 561
High level languages (HLLs) 572
High noise margin 559–560
High strobes 561
Higher levels of abstraction 587
High-level languages (HLLs) 365
High-resistance leakage 302
High-speed functional tester 286
History file 615, 623
Hi-TEA (High-Level Test Economics Advisor) 25
Hold time 43, 238, 424
Homing sequences 267
Horizontal lists 65
Hot spots 365
Huffman model 39–40, 53

Hyperactive fault 147
Hypertrophic fault 147

I

ICT See In-circuit tester
IDDQ
  coverage 556
  current drain 551
  current flow 551
  design rules 553
  empirical selection of threshold 557
  fault simulation 555, 560
  histogram of IDDQ current 556, 557
  measuring current flow 557–559
  monitoring 551
  pullups/pulldowns forbidden 553
  threshold for IDDQ, choosing 556–557
  threshold voltage Vt 562–563
Identity matrix 541
IEEE 1149.1 boundary scan 302–303, 434–442
IEEE-P1450 *See* Standard Test Interface Language (STIL)
Image mode, E-beam 300
Immediate dominator 204
Imminent range 65–66
Implementation-free 86
Implementing the LFSR 459–460
Implication 167–168, 202, 369
Implication tables 595
Imply Operation 369–370
Improving controllability and observability 418
Improvement in memory reliability 543–545
In-circuit tester (ICT) 302–304, 307–310, 389, 434–435
Incoming inspection 302–303
Incremental fault simulation 349
Incremental improvement in fault coverage 556
Indefinite paths 85
Indefinite state 83
Indeterminate state 48, 122
Indeterminate Value (X) 234–235
Indirectably observable 613
Indistinguishable blocks 490–493
Infix notation 637
Infrared thermography 317
Inhibit D-cubes 253
Inhibit memory control signals 419
Initial conditions 605
Initial objective(s) (IO) 190, 195, 198–199
Initial state 584
Initialization mode 623
Initialization problem 237
Initialization sequence 253, 259
Initialization stimuli 609
Injecting bugs 581
Inject fault symptoms 486
Injected errors 586
Inner product of two vectors 538
Input difference event sources 150
Input fault origin (IFO) 143–145
Input-bridging fault 335
Instruction retry 486
Integrated circuit(s) (IC) 2, 33–34, 120
Intellectual property (IP) 35, 299, 451
Interdependent goals 620
Intermittent faults 486
Internally balanced acyclic sequential circuit 263
Intersection of singular cubes 172
Intersection of fault lists 151
Intersection rule(s) 254, 257
Intrinsic weight 242–243
Irreducible polynomials 457, 499
Irredundant 334
Iterative array 241
Iterative Fault Simulation 348–349
Iterative test generator (ITG) 241–246
ITTAP 404

J

JK flip-flop 41–43, 249, 596
JTAG (Joint Test Action Group) See IEEE 1149.1
Jumper wires 395
Justification 168, 593

K

Karnaugh map(s) 176
Keating-Meyer circuit 557
Knowledge base 608, 629–630
Known good board (KGB) 312–313
Known good die (KGD) 24

L

Large-scale integration (LSI) 34, 388
LASAR 158, 205
Last-in, first-out (LIFO) stack 197
Lattice 77
LBIST (logic BIST) 479
Leakage current 298, 553, 556
Leakage path 552
Learn mode 609, 630–633
Learning curve 572
Learning phase 202
Least common multiple 501–502
Least fixed point 80
Least upper bound (lub) 77, 80
Level of a logic element 45
Level-sensitive flip-flops 42–43
Level sensitive scan design (LSSD) 412–417, 474–476
  A clock 412, 415
  B clock 412, 414–415
  C clock 412–415
  L1 latch 412–415
  L2 latch 412–415
  design rules 414
Levelized logic 45
LFSR See Linear-feedback shift-register
Libraries of tests 309
Library of parameterized models (LPM) 589–593, 615, 631
LIFO (last-in, first-out) memory 514
Limited *n*-level search 599
Linear associative algebra over F 457
Linear linked list 64
Linear span of *S* 539
Linear-feedback-shift-register (LFSR) 454–451, 459–462, 465, 467, 470–472, 475, 477, 481–483
Linearly independent 539
Lint 569
Liveness properties 641–642
Loading the scan chains 423, 453
Lockup latch 423–424
Logic contention 553
Lookahead strategy 611
Lookup tables 71
Loop unrolling 157
Loop-cutting algorithm 241, 263
Loop-free 260, 427
Low frequency (LF) set 431
Low strobes 561
LSSD See Level sensitive scan design

M

Macroblock(s) 490–493
Macrocells 326
Macrolan (Medium Access Controller) 480–482
Maintenance processor 484–485
Management information file (MIF) 487
Manufacturing faults 337, 362
Manufacturing management system (MMS) 302, 304
Manufacturing test 120, 301–304, 567
Map file 607–608, 612–613, 638
Master fault file 350, 368
Mathematical Basis For Self-Test 455–458
MaxGoal strategy 627–628
MaxGoal versus MinGoal 627
Maximize fault comprehension 373
Maximum ambiguity 267–268
Maximum comprehension 353
Maximum fault coverage 371
Maximum likelihood decoding 541
Maximum number of simulation steps 76
Maximum resolution 353
Maxterm 89
Mean time between failure(s) (MTBF) 28, 444, 545
Mean time to repair (MTTR) 29, 444
Mean-time-to-failure (MTTF) 503
Measuring Simulation Thoroughness 575–581
Medium-scale integration (MSI) 34, 388
Memory access time 537
Memory array faults 522
Memory behind tester channels 422
Memory built-in-self-test (MBIST) 524
Memory cell faults 530
Memory management 147
Memory organization, 2-D 515
Memory faults
  address nonuniqueness 521-522
  cell opens 521
  cell/column/row disturb 521
  data sensitivity 521
  disturb sensitivity 522

read/write logic faults 522
refresh sensitivity 521–522
sense amplifier interaction 521
slow access time 521
slow write recovery 521
static data losses 521
Memory test
13N algorithm 529–531
9N algorithm 529–531
address test 520
all 0s 517
all 1s 517
checkerboard test 519
dynamic test 517
functional test 517
galloping diagonal 519
GALPAT 517–519, 524–529
march test 519
march pattern 533
moving Inversions test 520
ping-pong test 517, 529
read disturb test 535
sliding diagonal 519
surround-by-complement (SBC) 395
surround read disturb 520
surround write disturb 520
walking pattern 519, 529
write Recovery 520
Merge fault 381
Merge node 496
Metal oxide semiconductor (MOS) 36, 338
Microblock(s) 489–493
Microcode 496
Microprocessor Matrix 493–494
Mimicing behavior of the human engineer 615
MinGoal strategy 627–628
Minimal test set 375
Min-Max timing 72–74
Minterm 89
Misaligned masks 521
MISR *See* Multiple-input shift register
Mode control 408
Model 3, 33
Model checking 640–648
Modular decomposition 36
Monostable 272, 391
Multichip logic module (MLM) 474–476
Multi-chip modules (MCM) 23–24
Multiple access fault 530
Multiple-array multiple bit (MAMB) 532
Multiple-array single bit (MASB) 532
Multiple backtrace 193–194, 196–199, 202, 204–205
Multiple clock domains 412, 426
Multiple faults 464
Multiple sensitization states 617
Multiple-fault simulation 136
Multiple-input shift register (MISR) 455, 460–463, 475–478, 532
Multiple-Valued Simulation 61–64
Multiplexing Address and Data-in 418
Multiplication of scalar and vector 538
Multiplicative identity 457
Murphy's Model 12
Mutual exclusion (mutex) problem 643
Mutually exclusive 624

N

NAND latch 41
NAND Tree 433–434
Necessary assignment 206
Negative binomial distribution 13
Negative clock edge (Negedge) 54, 422, 590
Nine-value algebra 246–249
Nine-valued simulation 57–58
NMOS device 39
Nominal delay simulation 59–61, 69–70
Non-blocking assignment 580
Noncontrolling value (NCV) 468
Noncritical assignment 206, 210
Noncritical region 643
Nondestructive readout (NDRO) 515
Non-integral event timing 65
Non-recurring costs 21
Non-repeating sequence 459
Non-return format 295
Non-scan flip-flops 430
Nonvolatile memory 515
NOR latch 40
Normal distribution 13
N-stage counter 454–459
Null space of V 539
Number of device inputs (NDI) 468

O

Obscured functionality 609
Observability equations 388, 397, 399–403, 551, 576

Observability points 466
Observability relation 490
Observability tree 393
Off-path side effect 222–223
One-controllability 153
One-hot encoding 408, 499
On-path side effect 222
Operation in a degraded mode 486
Order of a polynomial 501
Ordered BDDs (OBDDs) *See* Reduced ordered BDDs
Ordered *n*-tuple 3
Ordering Relation 489–493
Ordering the scan-flops 425
Orthogonal vectors 538
Oscillations 49, 135, 235–236
Oscillator 390
Output fault origin (OFO) 145
Output leakage test 298
Overall test length 614
Overshoot 286

P

Package test 561
Pad limited die 407, 546
Parallel drivers 392
Parallel fault simulator 134–136, 155
Parallel load 408
Parallel pattern single fault propagation (PPSFP) 137–139, 155
Parallel value list (PV) 156
Parallelize operation 421
Parametric faults 238, 303
Parametric measurement unit (PMU) 298
Pareto chart(s) 305, 546
Parity bit(s) 392, 496, 545
Parity checker(s) 188, 486
Parity generator H 543
Parity matrix P 541
Parity tree 393
Parse tree 642
Partial BIST 482–483
Partial scan 426–432
  benefits of 427
  choosing scan-flops 430
  destructive partial-scan 428
  drawback to partial-scan 426
Partially symmetric 130
Partitioning into layers 490
Partitioning circuit(s) 464, 481
Passes, no. of fault simulation 147
Pass-fail vector 351
Passive fault tolerance 495
Path coverage 366, 576
Path enumeration 107
Path quantifiers
  existential 641
  universal 641
PatternBurst block 292–293
PatternExec block 292
Pause test 535
PDCF *See* Primitive D-Cube of Failure
Performance Enhancements, simulation 570–571
Performance monitoring 485, 496–498
Periodic maintenance 504
Peripheral component interconnect (PCI) 513
Permuting the critical 0 206
Personal computer (PC) 453–454, 484, 487, 533
Pesticide paradox 579
Petri net 602–607
Phase-locked loop (PLL) 479
Physical probing 299
Pin electronics 287–288, 311
Pin map 288, 315
Pin memory 285
Pitfalls When Building Goal Trees 626–627
PMOS device 39
PODEM (path oriented decision making) 188–194, 202, 205, 430, 614
Point accelerators 571, 579
Point-to-point continuity 302
Poisson distribution 12
Posedge 54, 590
Positive and negative edge clocking 423
Post burn-in check (PBIC) 561
Power consumption 364
Power management feature 633
Power margining 486
PPSFP *See* Parallel pattern single fault propagation
Predecessor(s) 45, 138–139, 156, 242
Predicate logic 637
Prefix notation 637
Preprocess mode 406

Preset distinguishing sequence 268
Preset experiments 267
Previous time frame (PTF) 251–252
Prime implicant 51, 173
Primitive D-cubes of failure (PDCF) 174–177, 180, 588, 592
Primitive element 174, 391, 397
Primitive function test pattern(s) (PFTP) 592, 615
Primitive polynomial 458–459, 470
Printed circuit board(s) (PCBs) 4, 33, 388
Probability distribution function 11–12
Probable detected faults 129, 236, 349, 363, 372
Procedure 182
Process yield 388
Product-of-sums 39
Profiler 350, 571
Program instructions 496
Programmable logic arrays (PLAs) 336–337
Programmable read-only memories (PROMs) 515
Programming element 535
Programming language interface (PLI) 366, 575
Propagate a trapped fault 627
Propagate faults 614
Propagation 167, 593
Propagation D-cube(s) 177–178, 181–182, 399, 597
Propagation search 598–599
Proposition 636
Propositional logic 216
Prototype 34–35
Proximity of cells to one another 522
Pseudo-combinational circuit 49
Pseudo-input(s) 49, 135, 241, 244–246
Pseudo-output(s) 49, 135, 241, 244–245, 247
Pseudo-random generator (PRG) 454, 475–476
Pseudo-random test program generator 583
Pseudo-random vectors 156
Pulse generator(s) 272, 390
Pure functional mode 629

Q

Quality 2
Quasiexhaustive test 482
QuiC-Mon circuit 558–559
Quiescent current 562
Quiescent current drain 553
Quiescent periods 553
Quiet vectors 553
Quietest method 554-556
Quotient polynomial 457

R

Race Detection 71–72
Race(s) 50, 132, 239, 312
Random access memory (RAM) 514
Random access scan 411
Random logic 535
Random pattern effectiveness 464
Random pattern resistant faults 467
Random patterns 342–343
Random sample 14
Random stimulus generation 581–587
Random test pattern generation (RTPG) 582, 586–587
Random test socket (RTS) 474–475
Random tester 587
Random-resistant 343
Rank-order 45, 47, 102–103, 106, 138, 184, 191, 203, 262, 371, 401, 570–571, 616
Reachability analysis 645
Reactive circuits 647
READ array 47–48
Receive list (RLIST) 145
Reconvergent Path 170
Record of successes 615
Recurring costs 20
Reduced ordered BDDs (ROBDD) 94, 219–220, 638–639, 645–647
Reduction properties 492
Redundant fault 334–335
Redundant logic 335, 553
Reflow process 317
Register transfer level (RTL) 36, 325
  circuit image 588–589
  models 146, 568
Regular structure 356
Reject rate 15, 17
Reject ratio 15, 18
Relative conductance 76
Reliability Improvements 543–545
Reliability of the system 504

Reliability problems 552
Remainder polynomial 457
Remote Procedure Calls (RPC) 487
Remote range 65–66
Remote test 484–488
Repair station 304, 314, 474
Repairable Memories 535–537
Replacement board kits 474
Reporter 350
Requirements analysis 6,8
Requirements errors 650
Residue class 457
Resistance ratios 75
Resolution, definition of 285
Resolution function, VHDL 62
Resolution of the diagnostics 478
Response learning 316
Return on investment (ROI) 357
Return-to-complement 295
Return-to-high-impedance 295
Return-to-one 295
Return-to-zero 295
Re-verify functionality and timing 420
Ringing 286
Ripple technique 297
Rise time 60
ROBDD See Reduced ordered BDD
Roll back the state 633
Root cause 623, 634
Root of polynomial 457–458
Row decoder 515
Row or column failure 544
Rows represent functional units 494
RTL See Register transfer level
Rule-based system 631
Rule-of-ten 23, 302, 443, 650

S

Safety properties 641–642
SAMB Single-array multiple bit 532
Sampling ICs 388
SASB Single-array single-bit 532
Satisfying these goals 618
Statistical bin limits (SBL) 560
Scalars 538
Scan chains, partitioning 425
Scan chains of unequal length 425
Scan Compliance 416–418
Scan mode 408
Scan path 407, 426
  implementing scan path 420–426
  multiple scan paths 421–422
  ordering elements in the scan path 420
  violations of scan rules 415–417
Scan test 477
Scan/Set flip-flops 430
Scan-flops 409–410, 421–422, 425–431, 480
Scanning electron microscope (SEM) 299–301
Schedule marker 66–67
Scheduler for nominal delay simulation 64–67
Scheduler, First-in first-out (FIFO) 56
Scheduling process 68
Schmoo plots 294–295, 311
SCIRTSS (Sequential CIRcuit Test Search System) 597–607, 617
SCOAP (Sandia Controllability Observability Analysis Program) 176, 410, 415–416, 441, 447, 484, 617, 633, 641, 644
Screen at sort 562
Search heuristics 632–633
SEC-DEC code 545
Second-degree hardcore 490
Seed's Model 12
Seeding of design errors 581
Selective trace 404
Selector block 291
Self modifying methods 599
Self-Checking Circuits 498–499
Self-initializing sequence 242, 251, 372
Self-learning 302
Self-masking 209
Self-Monitoring, Analysis and Reporting Technology (SMART) 488
Self-resetting flip-flop 390–391, 416
Self-Test Using Multiple Parallel Signatures (STUMPS) 474–480
  controller chip 477
  overhead for 476
Sensitive input 208
Sensitivity, definition of 285
Sensitivity list 54, 136
Sensitivity value 376
Sensitization requirements 605
Sensitization search 598–599, 601–602

Sensitization state 598
Sensitization strategy 600
Sensitize fault(s) 151, 614, 627
Sensitized path 165–170, 180, 182–184
Sensitizing state 617
Sequential Circuit Test Search System (SCIRTSS) 597–602
Sequential conflicts In goal trees 618–620
Sequential controllability 398
Sequential D-chains 253
Sequential depth 262, 430, 432
Sequential logic test complexity 259–260
Sequential Path Sensitizer (SPS) 252–259
Sequential test pattern generation 611
Serial access memories 514
Serial data compression 462
Serial Data Out 411
Serial fault simulation 134, 157
Serial/parallel shift register 594
Service layer 487
Seshu's Heuristics 239–241
  best next or return to good 240
  combinational 240
  reset 241
  wander 240
Setup time 43, 238
S-graph 261, 431–432
Shadow logic 418
Shannon's expansion 88, 92, 97–98
Shared resource tester 287
Shift-register latch (SRL) 412–415
Shorter channel 563
Signal strengths 61
Signature analysis 453–455, 459–464, 470–474
Signature, compressed 453
Signatured instruction stream 496
Simulator Oriented Fault Test Generator (SOFTG) 369
Simultaneous self-test (SST) 475
Single bit error 541
Single instruction, multiple-data (SIMD) 498
Single shots 390
Single-fault assumption 127, 136, 166, 177
Skew lot 561
Skew parameters 561
Slack 108–109
Slew rate 286
Small-scale integration (SSI) 33, 388
Socrates test pattern generator 202–205, 218
Soft core(s) 299, 451
Soft errors 537–538, 544
Software implemented fault tolerance (SIFT) 505
Soft IP 650
Software profiling 157
Solder reflow 317, 433
Spare column replacement 537
Spec block 291
Specification 568
Speed binning 284, 302
Spike 50
Spotting testability issues 362
SRAM 534
SRE (Spare Row Enable) 535
SRL See Shift register latch
Standard cell libraries 353
Standard cells 326
Standard Test Interface Language (STIL) 288–293
State point(s) 45, 639, 649
State search routines 618
State table 40–41, 265–267, 269–273
State transition graph 597
  nondeterministic state transition graph 643
State Traversal Problem 597
States applied analysis 137, 155–156
State–space search algorithms 599
State machine
  completely specified state machine (CSSM) 407
  finite state machine 39–40, 236
  incompletely specified state machine (ISSM) 408, 579
  muxed 407
  unused states in 392
Static analysis 569,627, 629
Static partitioning 79
Static RAM(s) (SRAMs) 77, 515
Static tester 284–286
Statistical bias 103
Statistical bin limits (SBL) 562
Statistical fault analysis (STAFAN) 152–154
Statistical fault sampling 156
Steady state signals 80–83
Stem of a net 332
Stem fault 332
Stimulus bypass 54, 136

Stimulus ordering 103
Stopping rule 582
Storage node 75, 80–82, 84
Stream of instructions 496
Stress logic components 486
Stretch-and-shrink 297–298
Strobe placement 294
Strobe-to-strobe variability 561
Strobe_width value 294
Strongly balanced acyclic circuit 263
Strongly connected component (SCC) 242–244
Structural model 131, 568
Structural tester 306
Stuck-At Fault(s) 125–127, 166, 464–465, 579
Stuck-fault metric 577
Stuck-open faults 334, 339–340
Stuck-to-neighbor 357
Subordinate goal 620
Subscripted D-algorithm 184–188, 371
Substitution of a row or column 535
Successor states 269, 273–274
Successors of net *m* 242
Sugar 648
Sum-of-products 39
Super flip-flop(s) 252–256, 258–259
Super logic block D-cubes 253
Switch-level
  blocked at node i 83
  model 36, 75
  simulation 74, 79
Switching matrix 287
Symbolic Model Verifier (SMV) 648
Symbolic simulation 636, 648–650
Synchronizing sequence 267, 269–271, 273–276
Synchronous circuit 40
Syndrome 541–542
System reconfiguration 484
System test 307
Systematic code 541
System-on-a-chip (SoC) 35, 299

T

T (Toggle) flip-flop 41
TAP Controller. See Test access port
Tape-out 635
Target fault 600
Target of opportunity 616
Targeting undetected faults 430
Taxonomy 102
TDX Supervisor 608, 619
Technology-Related Faults 337–339
Temporal assertion 577
Temporal operator(s) 640, 645
  eventually 640
  globally 640
  next 640
  until 640
Temporal sequence 640
Ternary algebra 52–54, 70
Ternary clause 216
Ternary extension 80
Ternary simulation 48, 61, 63, 70, 134
Test control logic 479
Test controller 484–486
Test cost(s) 20, 319
Test cost versus quality trade-offs 25
Test counting 374–378
Test data generation and management 453
Test Data Injection 498
Test Design Expert (TDX) 607–635
Test economics 20–23
Test effectiveness 14–15
Test Measure Effectiveness 405
Test pattern compaction 372–374, 425
  dynamic test pattern compaction 373–374
  static compaction 372–373
Test patterns 3
Test plan 315–316
Test Problems Caused By Sequential Logic 233–237
Test resistant logic 362
Test response compactor (TRC) 454, 459, 474
Test set reordering 425
Test transparency (TT) 19
Test vector ordering 234
Testability analysis 592
Testability analysis tools 426
Testability analyzer 607
Testability measures 405
Testable latches 417
Testdetect 182–184
Tester escape(s) 14, 18–20, 121, 131, 311, 341, 351, 579

Tester time 607
Tester-per-pin architecture 287, 294
Testing strategies 306
Theorem proving 636–637
Thermal conduction modules (TCM) 477
Thoroughness of the test program 388
Through-holes 302
Throughput 570
Time-domain reflectometry (TDR) 318
Timer test 483
Timescale 329
Time-to-market 7
Time-to-volume 387
Timing analysis 570
Timing block 291
Timing generator 287
Timing sets (TSETs) 294, 311, 345
Timing verification 106–110
Timing wheel 65, 103, 572
Toggle coverage 364–365, 553–554, 560, 567, 575
Topological path (TP) 250–251
Total ambiguity 267, 269
Total controllability and observability 395
Totally self-checking 498
Tox (oxide thickness) 563
Traffic light controller (TLC) 641–642
Transfer function for the QuiC-Mon circuit 558
Transients 309
Transistor conductances 81
Transistor network 74
Transition region 52
Transparent memory test 419
Trapped signal 78
Trapped electrical charge 339
Trapped fault(s) 368–369, 598, 601, 609–613, 615–616
  propagation 609
  selection 612
Traverse algorithm 93–94, 638
Triple modular redundancy (TMR) 503–505
Tri-state device 61, 128–129
Trying region 643
Tunneling current 562
Two pattern sequence 340

U

Unate function 130
Unate gates 331
Uncontrollable node 406
Undetectable fault(s) 405, 429
Unidirectional search 625
Unintended side effects 632
Unique address 537
Unique sensitization 194
Unique signal path 333
Unix socket 575
Unobservable regions 429
Unused logic 553
Unweighted successors 242–244
User defined primitive(s) (UDPs) 102, 146, 330
User-suggested trial vectors 602

V

Vector space(s) 456, 538–540
Venn diagram 324
Verification Interacting with Synthesis (VIS) 648
Verilog
  models 358–361, 517–519, 526–528, 626
  primitives 60, 78, 128
  testbench 572–573
Verilog-2001 572
VHDL (VHSIC Hardware Description Language) 7, 35, 60, 572–573
VHSIC (Very High Speed Integrated Circuit) 35
Virtual components (VC) 35
Visible fault effect 143
Visual Inspection 316–318
Volatile memory 514–515
Voltage contrast 300–301
Voter circuits 504

W

Wafer sort 4, 24, 307, 561
Watchdog timers 490
Wave formatter 286–287
Wave soldering 433
waveform mode 300
WaveformChar 291
WaveformTable entry 293

Weak signal 62
Weak write test mode (WWTM) 534–535
Weighted random patterns (WRP) 467–470, 479, 582
Weighted value WV 469
Weighting factor WF 469
WFC_LIST 292
White-box testing 568
Wire-gate 62–63
Witness 640
Word line 63–64
Writable control store (WCS) 485, 490
WRITE array 47–48
Write test data into memory 419
Write-only 612

X

X and *Y* address 411
X-generator 428

Y

Yield 2
  analysis 11–14
  crash 4
  enhancement 300

Z

Zero-controllability 153
Zero-delay simulator 105
ZOBI (zero hour burn-in) 560
ZOBI evaluation 562
Zoom table 71